Action Items:
• Assignment 2 is due tomorrow, midnight
• Assignment 3 is out
Announcement

- Programming Assignment 2 is due on this Friday, midnight
- Programming Assignment 3 is out
  - Trivia due on **Feb 20, noon**
  - Main assignment due on March 2, midnight
Today: Data Structures and Buffer Overflow

• Arrays
  • One-dimensional
  • Multi-dimensional (nested)

• Structures
  • Allocation
  • Access
  • Alignment

• Buffer Overflow
Array Allocation: Basic Principle

\[ T \ A[L]; \]

- Array of data type \( T \) and length \( L \)
- Contiguously allocated region of \( L \times \text{sizeof}(T) \) bytes in memory

```
char string[12];
int val[5];
double a[3];
char *p[3];
```
Array Access: Basic Principle

\[ T \ A[L]; \]

- Array of data type \( T \) and length \( L \)
- Identifier \( A \) can be used as a pointer to array element 0: Type \( T^* \)

```
int val[5];
```

<table>
<thead>
<tr>
<th>Reference</th>
<th>Type</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>val[4]</td>
<td>int</td>
<td>3</td>
</tr>
<tr>
<td>val</td>
<td>int *</td>
<td>x</td>
</tr>
<tr>
<td>val+1</td>
<td>int *</td>
<td>( x + 4 )</td>
</tr>
<tr>
<td>&amp;val[2]</td>
<td>int *</td>
<td>( x + 8 )</td>
</tr>
<tr>
<td>val[5]</td>
<td>int</td>
<td>??</td>
</tr>
<tr>
<td>*(val+1)</td>
<td>int</td>
<td>5</td>
</tr>
<tr>
<td>val + i</td>
<td>int *</td>
<td>( x + 4 \ i )</td>
</tr>
</tbody>
</table>
Multidimensional (Nested) Arrays
Multidimensional (Nested) Arrays

• Declaration

\[
T \ A[R][C] ;
\]

• 2D array of data type \( T \)

• \( R \) rows, \( C \) columns

• Type \( T \) element requires \( K \) bytes

\[
\begin{bmatrix}
A[0][0] & \cdots & A[0][C-1] \\
\vdots & & \vdots \\
A[R-1][0] & \cdots & A[R-1][C-1]
\end{bmatrix}
\]
Multidimensional (Nested) Arrays

• Declaration
  
  \[ T \ A[R][C]; \]
  
  • 2D array of data type \( T \)
  
  • \( R \) rows, \( C \) columns
  
  • Type \( T \) element requires \( K \) bytes

• Array Size
  
  • \( R \times C \times K \) bytes
Multidimensional (Nested) Arrays

- **Declaration**
  \[ T \ A[R][C]; \]
  - 2D array of data type \( T \)
  - \( R \) rows, \( C \) columns
  - Type \( T \) element requires \( K \) bytes

- **Array Size**
  - \( R \times C \times K \) bytes

- **Arrangement**
  - Row-Major Ordering in most languages, including C

```c
int A[R][C];
```
Nested Array Row Access

• \( T \ A[R][C]; \)
  • \( A[i] \) is array of \( C \) elements
  • Each element of type \( T \) requires \( K \) bytes
  • Starting address \( A + i \times (C \times K) \)

\[
\text{int } A[R][C];
\]
Nested Array Element Access

- Array Elements
  - $A[i][j]$ is element of type $T$, which requires $K$ bytes
  - Address $A + i \times (C \times K) + j \times K = A + (i \times C + j) \times K$

```c
int A[R][C];
```

```
\begin{array}{c}
  \text{A} \\
  \begin{array}{c}
    \text{A[0][0]} \\
    \ldots \\
    \text{A[0][C-1]} \\
  \end{array} \\
  \ldots \\
  \begin{array}{c}
    \text{A[i][0]} \\
    \ldots \\
    \text{A[i][C-1]} \\
  \end{array} \\
  \ldots \\
  \begin{array}{c}
    \text{A[R-1][0]} \\
    \ldots \\
    \text{A[R-1][C-1]} \\
  \end{array}
\end{array}
```

$A + (i \times C \times 4) + (j \times 4)$
Today: Data Structures and Buffer Overflow

• Arrays
  • One-dimensional
  • Multi-dimensional (nested)

• Structures
  • Allocation
  • Access
  • Alignment

• Buffer Overflow
Structures

struct rec {
    int a[4];
    size_t i;
    struct rec *next;
};

- Characteristics
  - Contiguously-allocated region of memory
  - Refer to members within struct by names
  - Members may be of different types
**Structures**

```c
struct rec {
    int a[4];
    size_t i;
    struct rec *next;
};
```

- **Accessing struct member**
  - Given a struct, we can use the . operator, just like in Java:
    ```c
    struct rec r1; r1.i = val;
    ```
  - What if we have a pointer to a struct: `struct rec* r = &r1`
    ```c
    Using * and . operators: (*r).i = val;
    Or simply, the -> operator for short: r->i = val;
    ```
Generating Pointer to Structure Member

```c
struct rec {
    int a[4];
    size_t i;
    struct rec *next;
};
```

Diagram:
- `r` refers to the memory address
- `r+4*idx` refers to the memory address after offset `idx` times 4
- The diagram shows the structure `rec` with fields `a`, `i`, and `next` at different memory locations.
Generating Pointer to Structure Member

```c
struct rec {
    int a[4];
    size_t i;
    struct rec *next;
};

int *get_ap
    (struct rec *r, size_t idx)
{
    return &r->a[idx];
}
```
Generating Pointer to Structure Member

```c
struct rec {
    int a[4];
    size_t i;
    struct rec *next;
};

int *get_ap (struct rec *r, size_t idx)
{
    return &r->a[idx];
}
```

Diagram:
```
<table>
<thead>
<tr>
<th>a</th>
<th>i</th>
<th>next</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>16</td>
<td>24</td>
</tr>
</tbody>
</table>
```

\[ r \rightarrow r + 4 \times idx \]

\&(r->a[idx])

\&((\*r).a[idx])
Generating Pointer to Structure Member

```c
struct rec {
    int a[4];
    size_t i;
    struct rec *next;
};

int *get_ap
    (struct rec *r, size_t idx)
{
    return &r->a[idx];
}

int *get_ap (struct rec *r, size_t idx)
{
    return &r->a[idx];
}

// r in %rdi, idx in %rsi
leaq (%rdi,%rsi,4), %rax
ret
```

```
int *get_ap (struct rec *r, size_t idx)
{
    return &r->a[idx];
}

int *get_ap (struct rec *r, size_t idx)
{
    return &r->a[idx];
}

// r in %rdi, idx in %rsi
leaq (%rdi,%rsi,4), %rax
ret
```
Alignment

```c
struct S1 {
    char c;
    int i[2];
    double v;
} *p;
```
Alignment

• Unaligned Data

```
struct S1 {
    char c;
    int i[2];
    double v;
} *p;
```
Alignment

• Unaligned Data

```
struct S1 {
    char c;
    int i[2];
    double v;
} *p;
```

• Aligned Data
  • If the data type requires $K$ bytes, address must be multiple of $K$
Alignment

• Unaligned Data

• Aligned Data
  • If the data type requires $K$ bytes, address must be multiple of $K$
Alignment

• Unaligned Data

```
struct S1 {
    char c;
    int i[2];
    double v;
} *p;
```

• Aligned Data
  • If the data type requires $K$ bytes, address must be multiple of $K$
Alignment

• Unaligned Data

```
struct S1 {
    char c;
    int i[2];
    double v;
} *p;
```

• Aligned Data

- If the data type requires $K$ bytes, address must be multiple of $K$
Alignment

• Unaligned Data

```
struct S1 {
    char c;
    int i[2];
    double v;
} *p;
```

• Aligned Data

• If the data type requires $K$ bytes, address must be multiple of $K$
Alignment

• Unaligned Data

\[
\begin{array}{cccc}
\text{c} & \text{i[0]} & \text{i[1]} & \text{v} \\
p & p+1 & p+5 & p+9 & p+17
\end{array}
\]

• Aligned Data

• If the data type requires \( K \) bytes, address must be multiple of \( K \)

\[
\begin{array}{cccccccc}
\text{c} & \text{3 bytes} & \text{i[0]} & \text{i[1]} & \text{4 bytes} \\
p+0 & p+4 & p+8 & p+16
\end{array}
\]

\[
\begin{array}{cccc}
\text{Multiple of 4} & \text{Multiple of 8} \\
\end{array}
\]

```c
struct S1 {
    char c;
    int i[2];
    double v;
} *p;
```
Alignment

- **Unaligned Data**
  - Unaligned data is denoted as:
    
    ```
    struct S1 {
        char c;
        int i[2];
        double v;
    } *p;
    ```

- **Aligned Data**
  - If the data type requires \( K \) bytes, address must be multiple of \( K \)

```plaintext
\begin{align*}
    &c &i[0] &i[1] &v \\
    &p &p+1 &p+5 &p+9 &p+17
\end{align*}
```

```plaintext
\begin{align*}
    &c &3\text{ bytes} &i[0] &i[1] &4\text{ bytes} &v \\
    &p+0 &p+4 &p+8 &p+16 &p+24
\end{align*}
```

- Multiple of 4
- Multiple of 8
- Multiple of 8
- Multiple of 8
Alignment Principles

• Aligned Data
  • If the data type requires K bytes, address must be multiple of K

• Required on some machines; advised on x86-64

• Motivation for Aligning Data: Performance
  • Inefficient to load or store datum that spans quad word boundaries
  • Virtual memory trickier when datum spans 2 pages (later…)
  • Some machines don’t even support unaligned memory access

• Compiler
  • Inserts gaps in structure to ensure correct alignment of fields
  • `sizeof()` returns the actual size of structs (i.e., including padding)
Specific Cases of Alignment (x86-64)

- **1 byte:** char, ...
  - no restrictions on address

- **2 bytes:** short, ...
  - lowest 1 bit of address must be $0_2$

- **4 bytes:** int, float, ...
  - lowest 2 bits of address must be $00_2$

- **8 bytes:** double, long, char *, ...
  - lowest 3 bits of address must be $000_2$
Satisfying Alignment with Structures
Satisfying Alignment with Structures

• Within structure:
  • Must satisfy each element’s alignment requirement
Satisfying Alignment with Structures

• Within structure:
  - Must satisfy each element’s alignment requirement

• Overall structure placement
  - Each structure has alignment requirement $K$
    - $K =$ Largest alignment of any element
  - Initial address & structure length must be multiples of $K$
  - WHY?!
Satisfying Alignment with Structures

• **Within structure:**
  - Must satisfy each element’s alignment requirement

• **Overall structure placement**
  - Each structure has alignment requirement \( K \)
    - \( K = \) Largest alignment of any element
  - Initial address & structure length must be multiples of \( K \)
  - WHY?!

```c
struct S2 {
    double v;
    int i[2];
    char c;
} *p;
```

```
\begin{align*}
\text{p+0} & \quad \text{v} \\
\text{p+8} & \quad \text{i[0]} \quad \text{i[1]} \\
\text{p+16} & \quad \text{c} \quad \text{7 bytes} \\
\text{p+24} & \\
\end{align*}
```

Multiple of \( K=8 \)
Saving Space

- Put large data types first in a Struct
- This is not something that a C compiler would do
  - But knowing low-level details empower a C programmer to write more efficient code

```c
struct S4 {
    char c;
    int i;
    char d;
} *p;
```

```c
struct S5 {
    int i;
    char c;
    char d;
} *p;
```
Arrays of Structures

- Overall structure length multiple of K
- Satisfy alignment requirement for every element

```c
struct S2 {
    double v;
    int i[2];
    char c;
} a[10];
```
Accessing Array Elements

```c
struct S3 {
    short i;
    float v;
    short j;
} a[10];
```
Accessing Array Elements

```c
struct S3 {
    short i;
    float v;
    short j;
} a[10];

short get_j(int idx) {
    return a[idx].j;
}
```
Accessing Array Elements

```c
struct S3 {
    short i;
    float v;
    short j;
} a[10];

short get_j(int idx) {
    return a[idx].j;
}
```
Accessing Array Elements

struct S3 {
    short i;
    float v;
    short j;
} a[10];

short get_j(int idx) {
    return a[idx].j;
}

a[0] • • • a[idx] • • • a+12*idx

a+0 a+12 a+12*idx

i 2 bytes v j 2 bytes

a+12*idx a+12*idx+8
Accessing Array Elements

```c
struct S3 {
    short i;
    float v;
    short j;
} a[10];

short get_j(int idx)
{
    return a[idx].j;
}

# %rdi = idx
leaq (%rdi,%rdi,2),%rax  # 3*idx
movzwl a+8(%rax,4),%eax
```

Diagram showing the access to `a[idx]` in memory: `a+0`, `a+12`, `a+12*idx`, `a+12*idx+2` for `i`, `a+12*idx+8` for `j`.
Today: Data Structures and Buffer Overflow

• Arrays
  • One-dimensional
  • Multi-dimensional (nested)

• Structures
  • Allocation
  • Access
  • Alignment

• Buffer Overflow
typedef struct {
    int a[2];
    double d;
} struct_t;

double fun(int i) {
    volatile struct_t s;
    s.d = 3.14;
    s.a[i] = 1073741824; /* Possibly out of bounds */
    return s.d;
}
typedef struct {
    int a[2];
    double d;
} struct_t;

double fun(int i) {
    volatile struct_t s;
    s.d = 3.14;
    s.a[i] = 1073741824; /* Possibly out of bounds */
    return s.d;
}
Memory Referencing Bug Example

typedef struct {
    int a[2];
    double d;
} struct_t;

double fun(int i) {
    volatile struct_t s;
    s.d = 3.14;
    s.a[i] = 1073741824; /* Possibly out of bounds */
    return s.d;
}
String Library Code

/* Get string from stdin */
char *gets(char *dest)
{
    int c = getchar();
    char *p = dest;
    while (c != EOF && c != '\n') {
        *p++ = c;
        c = getchar();
    }
    *p = '\0';
    return dest;
}
String Library Code

- Implementation of Unix function `gets()`
  - No way to specify limit on number of characters to read

```c
/* Get string from stdin */
char *gets(char *dest)
{
    int c = getchar();
    char *p = dest;
    while (c != EOF && c != '\n') {
        *p++ = c;
        c = getchar();
    }
    *p = '\0';
    return dest;
}
```
String Library Code

• Implementation of Unix function `gets()`
  • No way to specify limit on number of characters to read
• Similar problems with other library functions
  • `strcpy, strcat`: Copy strings of arbitrary length
  • `scanf, fscanf, sscanf`, when given `%s` conversion specification

```c
/* Get string from stdin */
char *gets(char *dest)
{
    int c = getchar();
    char *p = dest;
    while (c != EOF && c != '\n') {
        *p++ = c;
        c = getchar();
    }
    *p = '\0';
    return dest;
}
```
Vulnerable Buffer Code

```c
/* Echo Line */
void echo()
{
    char buf[4];  /* Way too small! */
    gets(buf);
    puts(buf);
}

void call_echo() {
    echo();
}
```
Vulnerable Buffer Code

/* Echo Line */
void echo()
{
    char buf[4]; /* Way too small! */
    gets(buf);
    puts(buf);
}

void call_echo()
{
    echo();
}

unix>./bufdemo-nsp
Type a string: 012345678901234567890123
012345678901234567890123
Vulnerable Buffer Code

```c
/* Echo Line */
void echo()
{
    char buf[4];  /* Way too small! */
    gets(buf);
    puts(buf);
}

void call_echo()
{
    echo();
}
```

```
unix>./bufdemo-nsp
Type a string: 012345678901234567890123
012345678901234567890123
```

```
unix>./bufdemo-nsp
Type a string: 0123456789012345678901234
Segmentation Fault
```
Buffer Overflow Stack Example

Before call to gets

Stack Frame for call_echo

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
</tr>
<tr>
<td>00</td>
<td>40</td>
<td>06</td>
<td>f6</td>
<td></td>
</tr>
</tbody>
</table>

Stack Frame for echo
20 bytes unused

[3] [2] [1] [0]

void echo()
{
    char buf[4];
    gets(buf);
    ...
}

echo:
    subq $24, %rsp
    movq %rsp, %rdi
    call gets
    ...

call_echo:
    ...

00 40 06 f6

4006f1: callq 4006cf <echo>
4006f6: add $0x8,%rsp

buf ← %rsp
Buffer Overflow Stack Example #1

After call to `gets`

```
void echo()
{
    char buf[4];
    gets(buf);
    ...
}
```

echo:
```
subq $24, %rsp
movq %rsp, %rdi
```

call `gets`
```
...
```
```
4006f1:  callq 4006cf <echo>
```
```
4006f6:  add $0x8,%rsp
```
```
...
```

```
buf ← %rsp
```

```
unix> ./bufdemo-nsp
Type a string: 01234567890123456789012
01234567890123456789012
```

Overflowed buffer, but did not corrupt state
Buffer Overflow Stack Example #2

After call to gets

Stack Frame for call_echo

| 00 | 00 | 00 | 00 |
| 00 | 40 | 00 | 34 |
| 33 | 32 | 31 | 30 |
| 39 | 38 | 37 | 36 |
| 35 | 34 | 33 | 32 |
| 31 | 30 | 39 | 38 |
| 37 | 36 | 35 | 34 |
| 33 | 32 | 31 | 30 |

void echo()
{
    char buf[4];
    gets(buf);
    ...
}

echo:
    subq $24, %rsp
    movq %rsp, %rdi
    call gets
    ...

call_echo:

... 4006f1: callq 4006cf <echo>
... 4006f6: add $0x8,%rsp
...

buf ← %rsp

unix>./bufdemo-nsp
Type a string:0123456789012345678901234
Segmentation Fault

Overflowed buffer, and corrupt return address
Buffer Overflow Stack Example #3

After call to gets

Stack Frame for `call_echo`

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
</tr>
<tr>
<td>00</td>
<td>40</td>
<td>06</td>
<td>00</td>
<td></td>
</tr>
<tr>
<td>33</td>
<td>32</td>
<td>31</td>
<td>30</td>
<td></td>
</tr>
<tr>
<td>39</td>
<td>38</td>
<td>37</td>
<td>36</td>
<td></td>
</tr>
<tr>
<td>35</td>
<td>34</td>
<td>33</td>
<td>32</td>
<td></td>
</tr>
<tr>
<td>31</td>
<td>30</td>
<td>39</td>
<td>38</td>
<td></td>
</tr>
<tr>
<td>37</td>
<td>36</td>
<td>35</td>
<td>34</td>
<td></td>
</tr>
<tr>
<td>33</td>
<td>32</td>
<td>31</td>
<td>30</td>
<td></td>
</tr>
</tbody>
</table>

`void echo()`

```c
{
    char buf[4];
    gets(buf);
    ...
}
```

`echo:`

```c
subq $24, %rsp
movq %rsp, %rdi
call gets
```

`call_echo:`

```c
...
4006f1: callq 4006cf <echo>
4006f6: add $0x8,%rsp
...
```

buf ← %rsp

unix>./bufdemo-nsp
Type a string:012345678901234567890123 012345678901234567890123

Overflowed buffer, corrupt return address, but program appears to still work!
Buffer Overflow Stack Example #4

After call to gets

Stack Frame for call_echo

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
</tr>
<tr>
<td>00</td>
<td>40</td>
<td>06</td>
<td>00</td>
</tr>
<tr>
<td>33</td>
<td>32</td>
<td>31</td>
<td>30</td>
</tr>
<tr>
<td>39</td>
<td>38</td>
<td>37</td>
<td>36</td>
</tr>
<tr>
<td>35</td>
<td>34</td>
<td>33</td>
<td>32</td>
</tr>
<tr>
<td>31</td>
<td>30</td>
<td>39</td>
<td>38</td>
</tr>
<tr>
<td>37</td>
<td>36</td>
<td>35</td>
<td>34</td>
</tr>
<tr>
<td>33</td>
<td>32</td>
<td>31</td>
<td>30</td>
</tr>
</tbody>
</table>

register_tm_clones:

```
. . .
400600: mov %rsp,%rbp
400603: mov %rax,%rdx
400606: shr $0x3f,%rdx
40060a: add %rdx,%rax
40060d: sar %rax
400610: jne 400614
400612: pop %rbp
400613: retq
```

buf ← %rsp

“Returns” to unrelated code
Could be code controlled by attackers!
Such problems are a BIG deal
Such problems are a BIG deal

• Generally called a “buffer overflow”
  • when exceeding the memory size allocated for an array
  • It’s the #1 technical cause of security vulnerabilities
  • #1 overall cause is social engineering / user ignorance
Such problems are a BIG deal

• Generally called a “buffer overflow”
  • when exceeding the memory size allocated for an array
  • It’s the #1 technical cause of security vulnerabilities
  • #1 overall cause is social engineering / user ignorance

• The original Internet worm (1988) exploits buffer overflow
  • Invaded 10% of the Internet
  • Robert Morris, the authors of the worm, was a graduate student at Cornell and was later prosecuted
Such problems are a BIG deal

Robert Tappan Morris

Robert Tappan Morris (born November 8, 1965) is an American computer scientist and entrepreneur. He is best known[^3] for creating the Morris Worm in 1988, considered the first computer worm on the Internet.[^4]

Morris was prosecuted for releasing the worm, and became the first person convicted under the then-new Computer Fraud and Abuse Act.[^2][^5] He went on to co-found the online store Vieweb, one of the first web-based applications[^6], and later the funding firm Y Combinator—both with Paul Graham. He later joined the faculty in the department of Electrical Engineering and Computer Science at the Massachusetts Institute of Technology, where he received tenure in 2006.[^7]
What to do about buffer overflow attacks

• Avoid overflow vulnerabilities
• Employ system-level protections
• Have compiler use “stack canaries”
1. Avoid Overflow Vulnerabilities in Code (!)

```c
/* Echo Line */
void echo()
{
    char buf[4];  /* Way too small! */
    fgets(buf, 4, stdin);
    puts(buf);
}
```

- For example, use library routines that limit string lengths
  - `fgets` instead of `gets`
  - `strncpy` instead of `strcpy`
  - Don’t use `scanf` with `%s` conversion specification
    - Use `fgets` to read the string
    - Or use `%ns` where `n` is a suitable integer
2. System-Level Protections can help

• Randomized stack offsets
  • At start of program, allocate random amount of space on stack
  • Shifts stack addresses for entire program
  • Makes it difficult for hacker to predict beginning of inserted code
2. System-Level Protections can help

- Nonexecutable code segments
  - In traditional x86, can mark region of memory as either “read-only” or “writeable”
  - Can execute anything readable
  - X86-64 added explicit “execute” permission
  - Stack marked as non-executable

Any attempt to execute this code will fail
3. Stack Canaries can help

- **Idea**
  - Place special value (“canary”) on stack just beyond buffer
  - Check for corruption before exiting function

- **GCC Implementation**
  - `-fstack-protector`
  - Now the default (disabled earlier)
3. Stack Canaries can help

• Idea
  • Place special value (“canary”) on stack just beyond buffer
  • Check for corruption before exiting function

• GCC Implementation
  • `-fstack-protector`
  • Now the default (disabled earlier)

```plaintext
unix>./bufdemo-sp
Type a string: 0123456
0123456

unix>./bufdemo-sp
Type a string: 01234567
*** stack smashing detected ***
```
Setting Up Canary

Before call to gets

<table>
<thead>
<tr>
<th>Stack Frame for call_echo</th>
</tr>
</thead>
<tbody>
<tr>
<td>Return Address (8 bytes)</td>
</tr>
<tr>
<td>Canary (8 bytes)</td>
</tr>
</tbody>
</table>

buf ← %rsp

echo:

```assembly
... movq %fs:40, %rax  # Get canary
movq %rax, 8(%rsp)  # Place on stack
xorl %eax, %eax    # Erase canary
...```

/* Echo Line */
void echo()
{
    char buf[4]; /* Way too small! */
    gets(buf);
    puts(buf);
}
Checking Canary

After call to gets

Stack Frame for call_echo

Return Address (8 bytes)

Canary (8 bytes)

00 36 35 34
33 32 31 30

Input: 0123456

buf ← %rsp

echo:

... 

movq 8(%rsp), %rax  # Retrieve from stack
xorq %fs:40, %rax  # Compare to canary
je .L6  # If same, OK
call __stack_chk_fail  # FAIL
.L6: ...