Buffer overflows

Stack layout when calling functions

• What do we do when we call a function?
  • What data need to be stored?
  • Where do they go?

• How do we return from a function?
  • What data need to be restored?
  • Where do they come from?

Code examples
Stack layout when calling functions

```
void func(char *arg1, int arg2, int arg3)
{
    char loc1[4]
    int loc2;
}
```

Arguments pushed in reverse order of code

Local variables pushed in the same order as they appear in the code

Accessing variables

```
void func(char *arg1, int arg2, int arg3)
{
    char loc1[4]
    int loc2;
    ...
    loc2++;
    ...
}
```

Q: Where is (this) `loc2`?
A: -8(%ebp)

Stack frame for this call to `func`

- I don’t know where `loc2` is,
- and I don’t know how many args
- but `loc2` is always 8B before “???”'s
### Notation

- **%ebp**: A memory address in a register
- **(%ebp)**: The value at memory address %ebp (like dereferencing a pointer)

```
pushl %ebp
movl %esp %ebp /* %ebp = %esp */
```

### Returning from functions

```c
int main()
{
    ...
    func("Hey", 10, -3);
    ...
}
```

**Q: How do we restore %ebp?**

1. Push %ebp before locals
2. Set %ebp to current %esp
3. Set %ebp to(%ebp) at return
Returning from functions

```c
int main()
{
    ... 
    func("Hey", 10, -3);
    ...  Q: How do we resume here?
}
```

The instructions themselves are in memory

```
0xffffffff
... 0x5bf mov %esp,%ebp
0x5be push %ebp
... 0x49b movl $0x804..,(%esp)
0x493 movl $0xa,0x4(%esp)
0x4a2 call <func>
0x4a7 mov $0x0,%eax
0x493 movl $0xa,0x4(%esp)
... %eip
```
# Returning from functions

```c
int main()
{
    ...
    func("Hey", 10, -3);
    ...
    Q: How do we resume here?
}
```

---

**Stack frame for this call to func**

Set `%eip` to 4(%ebp) at return

Push next `%eip` before call

---

## Stack and functions: Summary

**Calling function:**
1. **Push arguments** onto the stack (in reverse)
2. **Push the return address**, i.e., the address of the instruction you want run after control returns to you: `%eip`+something
3. **Jump to the function’s address**

**Called function:**
4. **Push the old frame pointer** onto the stack: `%ebp`
5. **Set frame pointer** `%ebp` to where the end of the stack is right now: `%esp`
6. **Push local variables** onto the stack; access them as offsets from `%ebp`

**Returning function:**
7. **Reset the previous stack frame**: `%ebp = (%ebp)`
8. **Jump back to return address**: `%eip = 4(%ebp)`
Buffer overflows from 10,000 ft

- **Buffer =**
  - Contiguous set of a given data type
  - Common in C
    - All strings are buffers of char's

- **Overflow =**
  - Put more into the buffer than it can hold

- Where does the extra data go?

- Well now that you’re experts in memory layouts…

A buffer overflow example

```c
void func(char *arg1)
{
    char buffer[4];
    strcpy(buffer, arg1);
    ...
}

int main()
{
    char *mystr = "AuthMe!";
    func(mystr);
    ...
}
```

**Upon return, sets %ebp to 0x0021654d**

```assembly
M e ! \0
```

<table>
<thead>
<tr>
<th>Auth</th>
<th>4d 65 21 00</th>
<th>%eip</th>
<th>&amp;arg1</th>
</tr>
</thead>
<tbody>
<tr>
<td>buffer</td>
<td>SEGFAULT (0x00216551)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```
A buffer overflow example

```c
void func(char *arg1)
{
    int authenticated = 0;
    char buffer[4];
    strcpy(buffer, arg1);
    if(authenticated) { ... }
}

int main()
{
    char *mystr = "AuthMe!";
    func(mystr);
    ...
}
```

Code still runs; user now ‘authenticated’

```
M e ! \0
```

<table>
<thead>
<tr>
<th>Auth</th>
<th>4d 65 21 00</th>
<th>ebp</th>
<th>eip</th>
<th>&amp;arg1</th>
</tr>
</thead>
<tbody>
<tr>
<td>buffer</td>
<td>authenticated</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

```c
void vulnerable()
{
    char buf[80];
    gets(buf);
}

void still_vulnerable()
{
    char *buf = malloc(80);
    gets(buf);
}
```
```c
void safe()
{
    char buf[80];
    fgets(buf, 64, stdin);
}

void safer()
{
    char buf[80];
    fgets(buf, sizeof(buf), stdin);
}
```

**User-supplied strings**

- In these examples, we were providing our own strings
- But they come from users in myriad ways
  - Text input
  - Packets
  - Environment variables
  - File input…
What’s the worst that could happen?

```c
void func(char *arg1) {
    char buffer[4];
    strcpy(buffer, arg1);
    ...
}
```

All ours!

`strcpy` will let you write as much as you want (til a ‘\0’)

What could you write to memory to wreak havoc?

Stack and functions: Summary

**Calling function:**
1. **Push arguments** onto the stack (in reverse)
2. **Push the return address**, i.e., the address of the instruction you want run after control returns to you: `%eip+something`
3. **Jump to the function’s address**

**Called function:**
4. **Push the old frame pointer** onto the stack: `%ebp`
5. **Set frame pointer** `%ebp` to where the end of the stack is right now: `%esp`
6. **Push local variables** onto the stack; access them as offsets from `%ebp`

**Returning function:**
7. **Reset the previous stack frame**: `%ebp = (%ebp)`
8. **Jump back to return address**: `%eip = 4(%ebp)`
gdb tutorial

Your new best friends

```plaintext
if
Show info about the current frame
(prev. frame, locals/args, %ebp/%eip)

ir
Show info about registers
(%eip, %ebp, %esp, etc.)

x/<n> <addr>
Examine <n> bytes of memory
starting at address <addr>

b <function>
s
Set a breakpoint at <function>
step through execution (into calls)
```
Buffer overflow

Can over-write other data (“AuthMe!”)

Can over-write the program’s control flow (%eip)

```
char loc1[4];
```

gets(loc1);
strcpy(loc1, <user input>);
memcpy(loc1, <user input>);
etc.

Input writes from low to high addresses