Today

• What happens when a program starts running?
• Address spaces
• Virtual memory
Let’s start a program

$ ./bufbomb -u tbergan

**Goal:** execute `main()` in `./bufbomb`

```c
int main(int argc, char *argv[]) {
    ...
}
```

**Where**

- `argc = 3`
- `argv[0] = "./bufbomb"`
- `argv[1] = "-u"`
- `argv[2] = "tbergan"`

The shell executes this code:

```c
excl("./bufbomb", "-u", "tbergan", NULL);
```

**How does exec() work?**
What happens on exec()?

Steps to exec:
1. Load program executable
2. Copy the args into memory
3. Setup the registers
4. Jump to main()

Goal: execute main() in ./bufbomb

int main(int argc, char *argv[]) {
    ...
}

Where
argc = 3
argv[0] = "./bufbomb"
argv[1] = "-u"
argv[2] = "tbergan"
Each process has its own address space

here is a pointer
p: 0x0041ab8fe023ecd5

p1 address space

0  2^{64} - 1

NOT the same

p2 address space

0  2^{64} - 1
Address spaces are virtual

Here is a pointer:

\[
p: 0x0041ab8fe023ecd5
\]

Address spaces are virtual, not the same as physical memory.
Virtual Address Spaces

here is a pointer
p: 0x0041ab8fe023ecd5

p1 address space

virtual address table

physical address

physical memory
Virtual Address Spaces

P₁ address space
- code
- heap
- stack

P₂ address space
- code
- heap
- stack

page table

physical memory

0

2^{64}-1
Virtual address translation

memory is divided into pages

virtual memory

virtual address

page table

Virtual Page # | Physical Page #
---|---
2 | 5

physical memory

physical address

Step 1: translate the page #
Step 2: translate the offset
Virtual address translation

**Virtual address**: 0x0041ab8fe023ecd5

**Physical address**: 5230abeab44cf cd5

**Page table**

<table>
<thead>
<tr>
<th>Virtual Page #</th>
<th>Physical Page #</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0041ab...</td>
<td>0x5230a...</td>
</tr>
</tbody>
</table>
Virtual address translation

<table>
<thead>
<tr>
<th>Virtual Page #</th>
<th>Physical Page #</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0041ab...</td>
<td>0x5230a...</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Physical Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>5230abeab44cf</td>
</tr>
<tr>
<td>5230abeab44cf</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Virtual Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>0041ab8fe023e</td>
</tr>
<tr>
<td>0041ab8fe023e</td>
</tr>
</tbody>
</table>
Virtual Address Spaces

Do you ever want to share memory across processes?
Virtual Address Spaces

Do you ever want to share memory across processes?
- yes! shared libraries!

P₁ address space
- code
- shared lib
- heap
- stack

P₂ address space
- code
- shared lib
- heap
- stack

physical memory
Shared Libraries

A shared library:
- think `printf()`: `*.so` on Linux, `*.dll` on Windows
- share code pages in multiple address spaces (saves space!)

Problem: can’t let $P_2$ overwrite to $P_1$’s code!
- solution: map pages *read-only*
Shared Libraries

P₁ address space
0 - code - shared lib - heap - stack

Physical memory

Page table

<table>
<thead>
<tr>
<th>Virtual Address</th>
<th>Physical Address</th>
<th>Protection Bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0041ab...</td>
<td></td>
<td>✘ writable</td>
</tr>
</tbody>
</table>

Pages mapped read-only

Page table

<table>
<thead>
<tr>
<th>Virtual Address</th>
<th>Physical Address</th>
<th>Protection Bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x07eff...</td>
<td></td>
<td>✘ writable</td>
</tr>
</tbody>
</table>

P₂ address space
0 - code - shared lib - heap - stack
Page table protection bits
(partial list)

• **writable** bit
  - is the page writable?
  - when unset, the page is *read-only*

  **Why would you want this?**
  - protect code pages (don’t accidentally overwrite)
  - read-only data (e.g. constant strings literals: “xyz”)

• **executable** bit
  - is the page executable?
  - when unset, code on the page *cannot* be executed

  **Why would you want this?**
  - protect non-code pages (e.g. prevents buffer overflow exploits)
  - read-only data (e.g. constant strings literals: “xyz”)
Shared Libraries

Shared libraries are loaded at runtime

New steps to start a program:
1. Load program executable
   1a. Load shared libraries
2. Copy the args into memory
3. Setup the registers
4. Jump to main()
**Shared Libraries**

How do we know the address of `memcpy`?
- it depends on where the lib was loaded
- solution: *jump table*

![P1 address space](image)

![P2 address space](image)
Shared Libraries

P₁ address space

0x0A0 call foo

0x105 foo:
call *jumpTable[42]

Jump table initially empty

jumpTable = {
  [0] = ?
  [1] = ?
  ...
  [42] = ?
  ...
}

Library call indirects through jump table
Shared Libraries

\( \mathbb{P}_1 \) address space

- code
- shared lib
- heap
- stack

\[
\begin{array}{c}
0x0A0 \text{ call foo} \\
0x105 \text{ foo:} \\
\text{ call *jumpTable[42]} \\
\end{array}
\]

jumpTable = {
  [0] = ?
  [1] = ?
  ... 
  [42] = &memcpy,
  ... 
}

\[
0x3FC \text{ memcpy:} \\
\]

Jump table fixed when library is loaded
- by a program called a loader