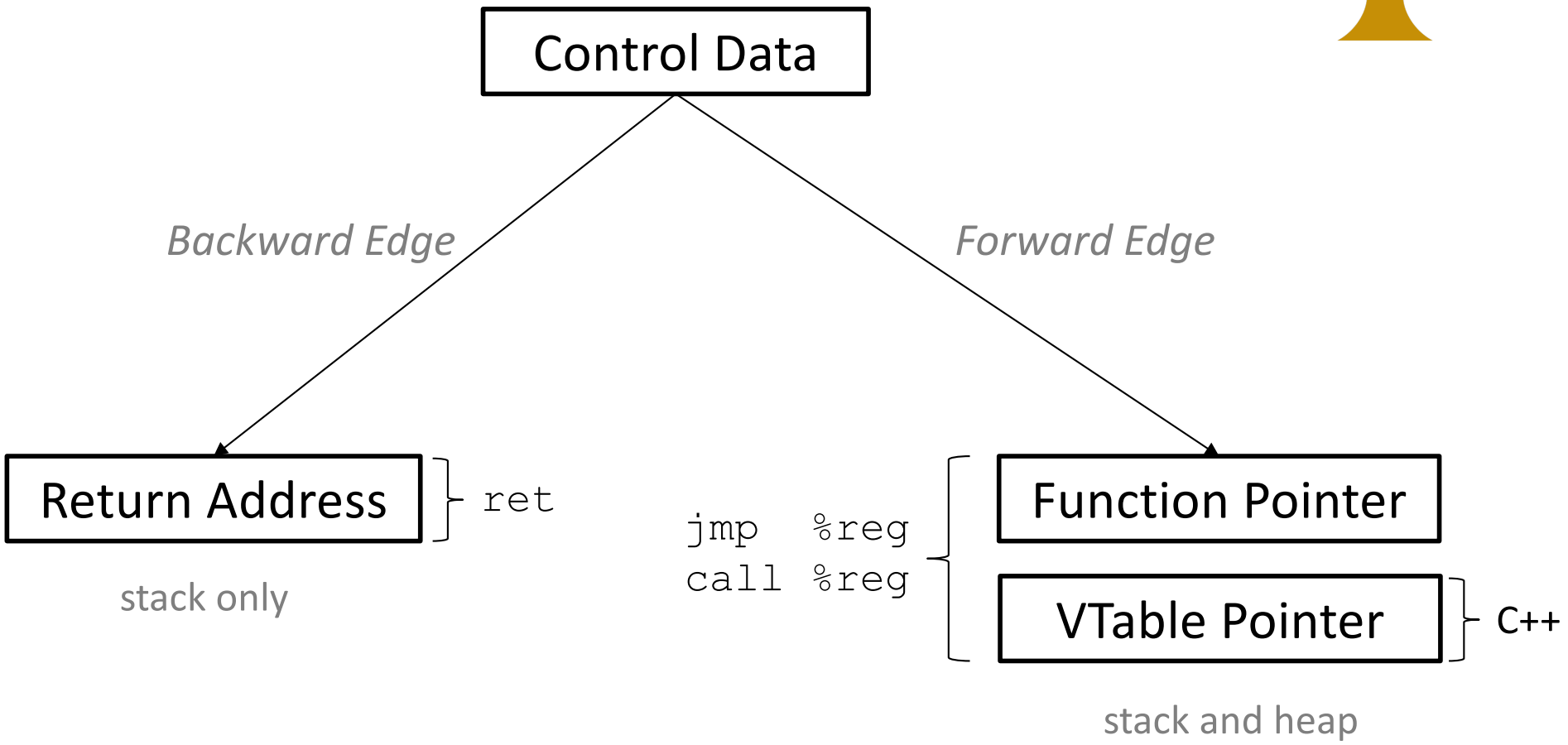
CS326 – Systems Security

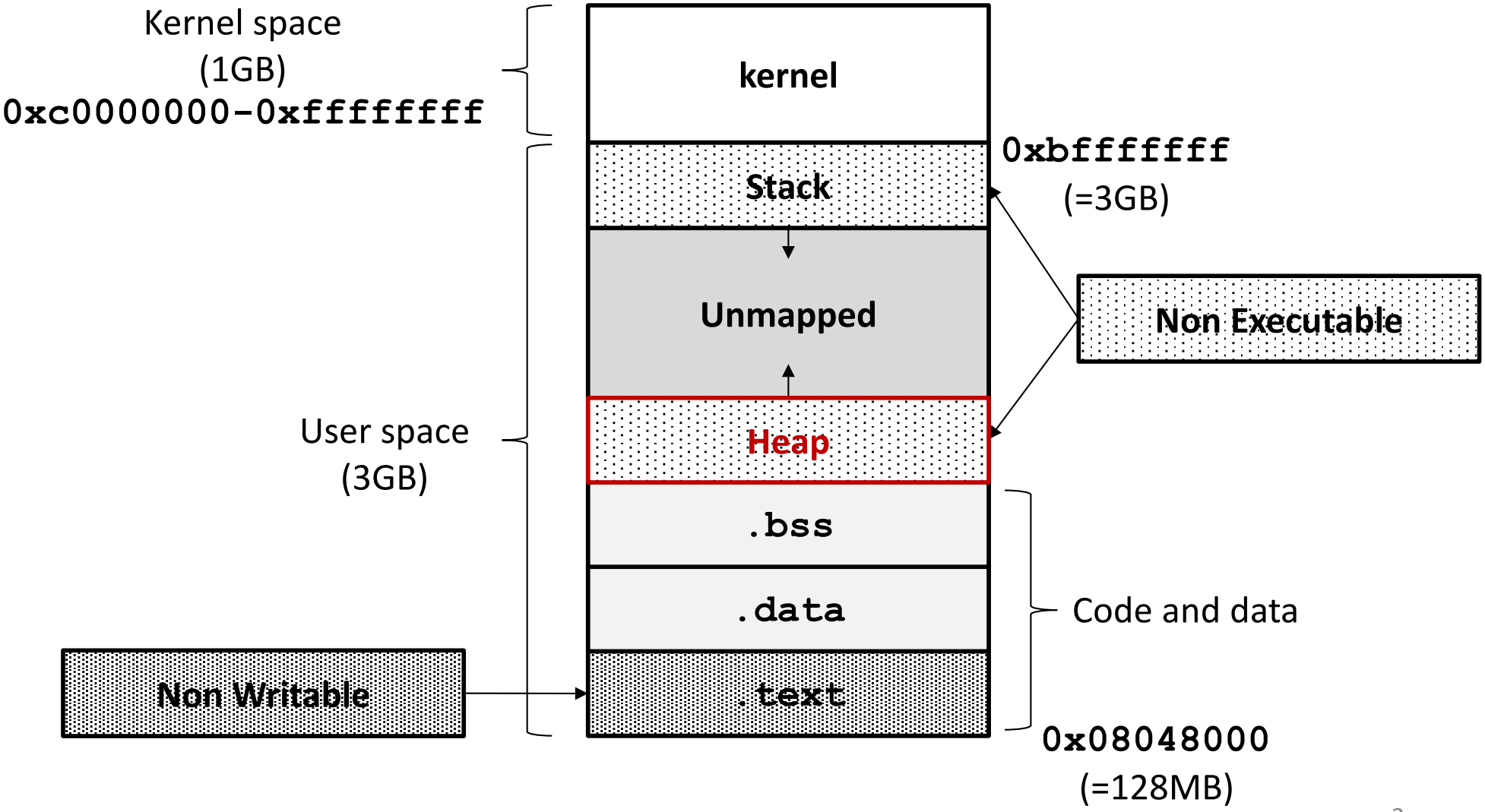
Lecture 15

Heap Exploitation

Elias Athanasopoulos
athanasopoulos.elias@ucy.ac.cy


Control Data
Backward Edge
Forward Edge
Return Address
ret
stack only
jmp %reg
call %reg
Function Pointer
VTable Pointer
C++
stack and heap

Process Memory Layout
32-bit, Linux (W^X)



Heap

- A memory area that stores resources dynamically
- Managed by a user-space library
  - Standard (libc), jemalloc, tcmalloc, etc.
- API access
  - `malloc()`, `free()`, `realloc()`, etc.
- No CPU support
  - Stack was accessed using `pop`, `push`, `call`, `ret`

Heap Overflows

- Overwrite control data in the heap
  - No return addresses in the heap
  - Function pointers, VTable pointers
- Much harder for exploit writing
  - ROP uses the stack for executing gadgets
  - Stack pointer used as a program counter
  - There is no stack in the heap
  - Executing a single gadget will transfer control to the stack when the first `ret` executes

Stack Pivoting

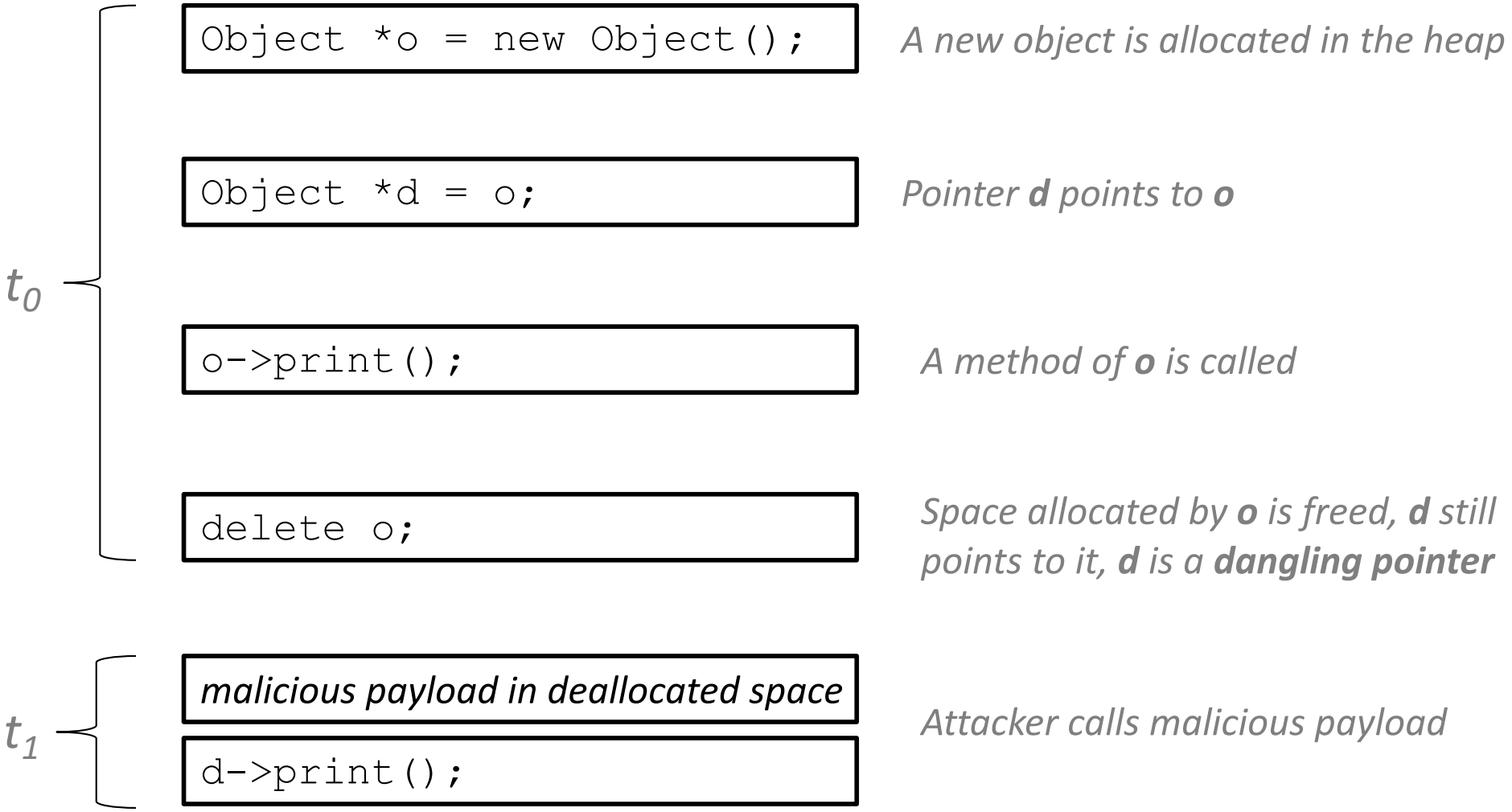
- The first gadget must create a virtual stack in the heap
  - Change the stack pointer to point in the heap
- Example gadget

```
xchg %eax, %esp
ret
```

Spatial vs Temporal Safety

- Spatial Safety
  - Overflow linearly a buffer and overwrite data stored next to it
  - Based on data geometry
  - Handy for stacks (data are stacked together)
- Temporal Safety
  - Abuse data access over time
  - Place malicious data in de-allocated space and trigger dangling pointers that still point to it

Use-after-free



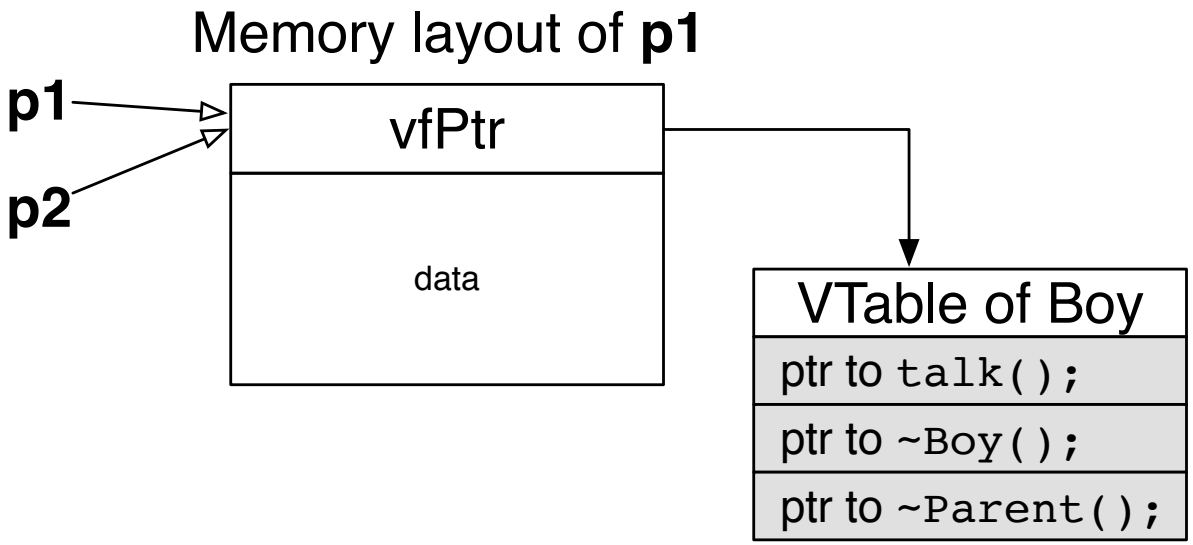
C++ Virtual Objects

```
class Parent {
public:
    virtual void talk();
};

class Boy: public Parent {
public:
    void talk();
};

class Girl: public Parent {
public:
    void talk();
};
```

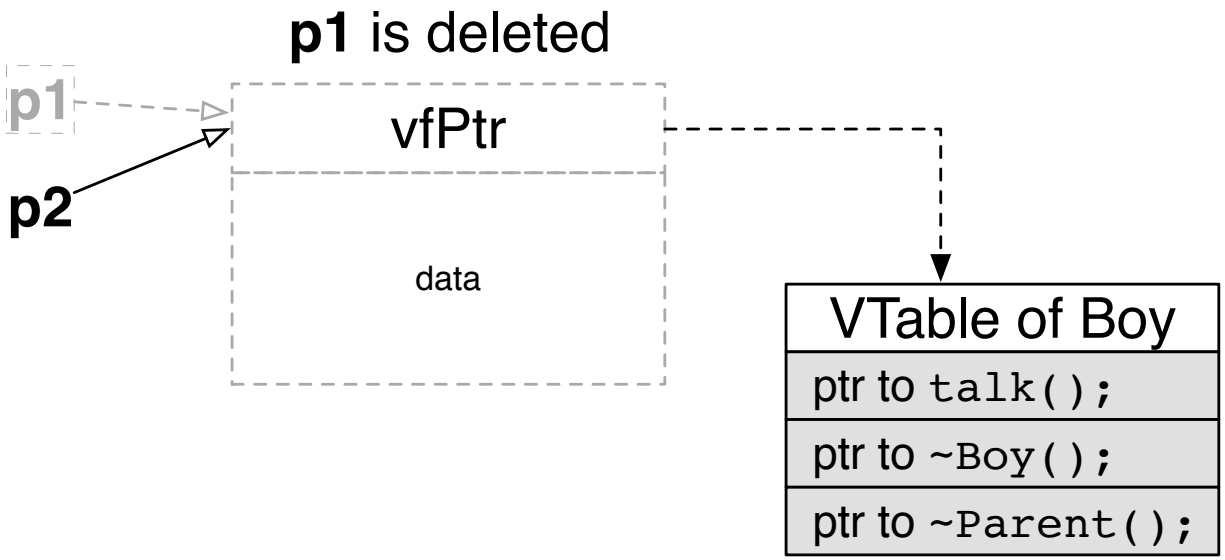
Use-after-free Bug

```
int main(int argc, char *argv[]) {
    Parent *p1, *p2;
    ...
    input == true ? p1 = new Boy() : p1 = new Girl();
    p1->talk();
    p2 = p1;
    /* Destructors (Boy or Girl, Parent) are called */
    delete p1;
    /* p2 is now dangling */
    ...
    /* use-after-free trigger */
    p2->talk();
    return 1;
}
```

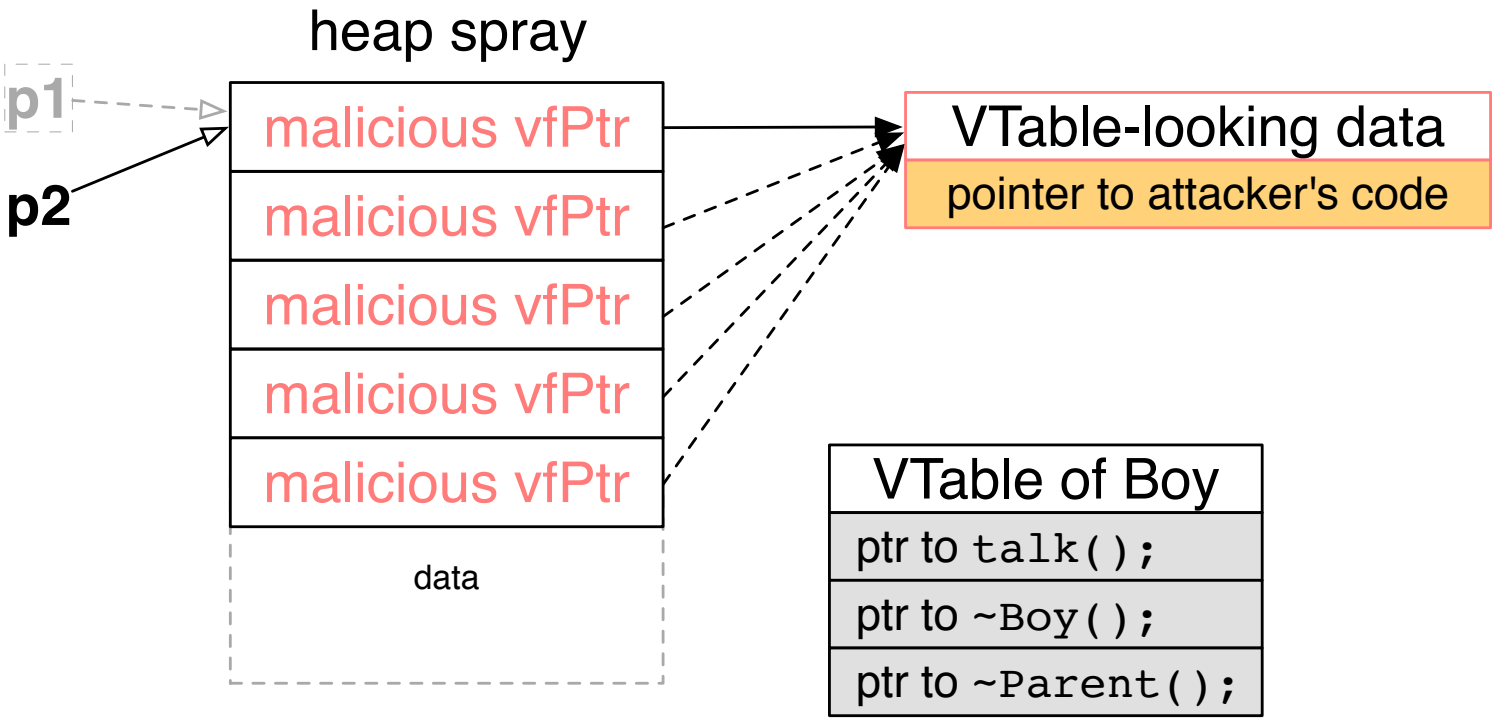
VTable Layout



Dangling Pointer



Heap Feng Shui



Integer Overflow

```
off_t j, pg_start = /* from user space */;
size_t i, page_count = ...;
int num_entries = ...;

if (pg_start + page_count > num_entries)
    return -EINVAL;
...
for (i = 0, j = pg_start; i < page_count; i++, j++)
    /* write to some address with offset j */;
```

An adversary can provide a large `pg_start` value from user space to bypass the check `pg_start + page_count > num_entries`, since `pg_start + page_count` wraps around. This leads to out-of-bounds memory writes in later code

Integer Overflow

```
off_t j, pg_start = /* from user space */;
size_t i, page_count = ...;
int num_entries = ...;

if ((pg_start + page_count > num_entries) ||
    (pg_start + page_count < pg_start))
    return -EINVAL;
...
for (i = 0, j = pg_start; i < page_count; i++, j++)
    /* write to some address with offset j */;
```