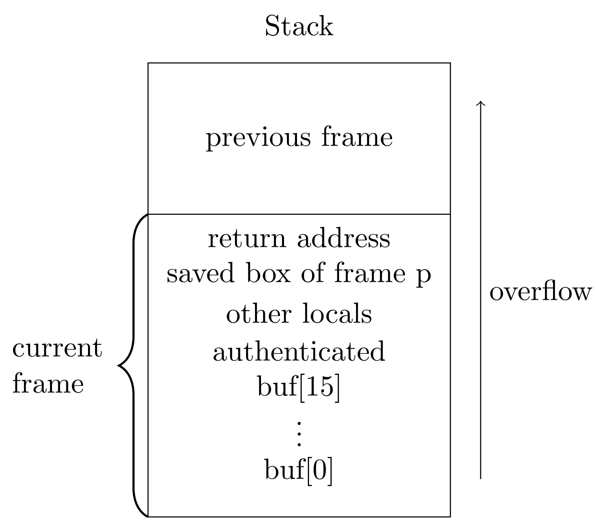
September 1 : Memory safety, Buffer Overflows attacks

Scribe: Katia Patkin

1 Requirments

- System software written in an unsafe language (C), exposes raw pointers to the developer.
- Architectural layout of data.

```
void login()
{
    char buf[16];
    int authenticated = 0;
    [other local var]
    gets(buf);
    ...
}
```



Attacker can:

- Set data (e.g. *authentication* bit).
- Get control of program flow, run with process privilege.

2 Types of Buffer Overflow

1. **Stack Smashing:** Attacker overwrites return address and points to attacker supplied code.
2. **Arc injection:** Attacker overwrite *return address* to points to existing code.

   Example: return to libc

```
void system(char * arg)
{
    check_validity(arg);
```

```
        R = arg;
        target:
                execl(R,...)
    }
```

Steps:

(a) Set return address to target.

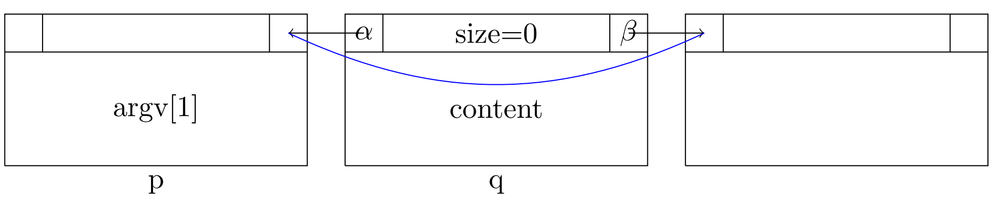
(b) Ensure R (system register) points to attacker code (based on vulnerable program logics registers are reused)

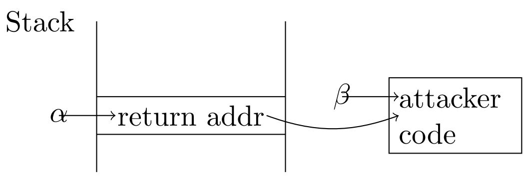
3. **Pointer subterfuge:** Attack exploiting pointer overwrite.

4. **Heap Smashing:**

```
int main(int argc, char* argv)
{
    p = malloc(1024);
    q = malloc(1024);
    strcpy(p, argv[1]);
    ...
}
```

Simplified heap model:





[*] Upon block free (size=0) heap manager sets previous pointer to the next pointer.

Steps:

(a) Overwrite heap block such that previous pointer (α) points to return address, next pointer (β) points to attacker's code and size=0.

(b) Heap manager frees block set location at α to point to β.

$\Rightarrow$ return address points to attacker code.

This attack is not very common, because the memory layout is less predictable and it is a more complicated attack.

3 Fixes

1. **Avoid bugs in C code:** Pros: solves in the sources. Cons: hard to write bug-free code.
2. **Build tools that help programmers find bugs:**

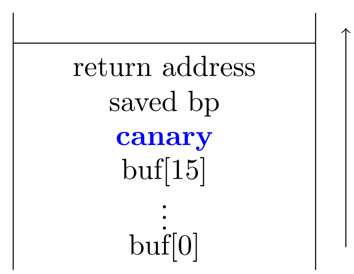
   Example:

   ```
   void foo(int* p)
    {
        int offset;
        int* z = p + offset;
               ...
    }
   ```

   Static checker: Checks that *offset* is not intialized. offset hence can get any value, which means pointer could point to anything. Cons: hard to find all bugs.
3. **Use a memory-safe language:**

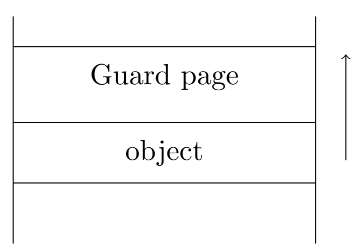
   Cons:
   - Not good for performance.
   - There is legacy code.
   - Not suitable for writing low-level code.
4. **Bounds checking:**
   - Canaries: Modifies source code

     Compiler places canary (random value) before local variables upon entry in function and checks before return.

   - Electric fences:

     Object is followed by a *guard page*. Any access to guard page triggers page fault.

     Cons: takes a lot of memory space, can be used for DoS attacks.
   - Baggy bounds:

     Goal: to check that the pointers are in range.

     Example:

```
char x[1024];
char* y = &x[107];
y+2124 ...
```

Check for pointer arithmetic that it is in bound.
How: For a pointer p' that is derived from p. p' should only be dereferenced to access memory that belongs to p.

4 Fat pointers

Each pointer holds bound information:

base	end	current address

Cons:

- Performance overhead: for every pointer dereference, check bounds.
- Memory overhead: every 32-bit pointer is now 96-bit pointer.
- Incompatible with existing binaries.

5 Baggy bounds

Use data structures to keep bounds of each pointer.

Interpose on two operations:

1. **pointer arithmetic:**

   ```
   char* q = p + 256
   ```

   Needed to check pointer provenance (which pointer it was derived from)

2. **pointer dereference:**

   ```
   char p[256];
   ```

   Needed because in arithmetic intermediate value might be out of bound.

Implementation:

1. Align and allocate in the power of 2. Ex.: $malloc(44) \rightarrow 64$.
2. Express size of pointer as $\log_2(alloc\ size)$.
3. Store pointer to size in a linear array.
4. Allocate memory at slot granularity (16 bytes for Baggy).

Example:
`p = malloc(16)` $\rightarrow$ *alloc size* $= 16, size = 4, slot = 1 \rightarrow$ `table[p\slot_size]=4`.
`p = malloc(44)` $\rightarrow$ *alloc size* $= 64, size = 6, slot = 4 \rightarrow$ `table[p\slot_size`$_0$`]=6`,
`...,` `table[p\slot_size`$_3$`]=6`

Check p' is in the bound of p:

C code:

```
p' = p + i
```

Bounds check:

```
size = 1 ≪ table[p ≫ log(slot size)]
base = p & (size -1)
base ≤ p' < base + size
```