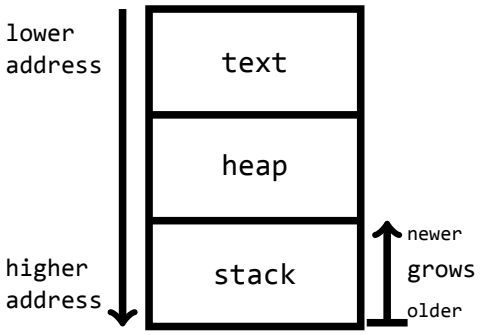
Handout 3 (Buffer Overflow Attacks)

By far the most popular attack method on computers are buffer overflow attacks or simple variations thereof. The popularity is unfortunate because we nowadays have technology in place to prevent them effectively. But these kind of attacks are still very relevant even today since there are many legacy systems out there and also many modern embedded systems do not take any precautions to prevent such attacks.

To understand how buffer overflow attacks work, we have to have a look at how computers work "under the hood" (on the machine level) and also understand some aspects of the C/C++ programming language. This might not be everyday fare for computer science students, but who said that criminal hackers restrict themselves to everyday fare? Not to mention the free-riding script-kiddies who use this technology without even knowing what the underlying ideas are.

For buffer overflow attacks to work, a number of innocent design decisions, which are really benign on their own, need to conspire against you. All these decisions were pretty much taken at a time when there was no Internet: C was introduced around 1973; the Internet TCP/IP protocol was standardised in 1982 by which time there were maybe 500 servers connected (and all users were well-behaved, mostly academics); Intel's first 8086 CPUs arrived around 1977. So nobody of the "forefathers" can really be blamed, but as mentioned above we should already be way beyond the point that buffer overflow attacks are worth a thought. Unfortunately, this is far from the truth. I let you think why?

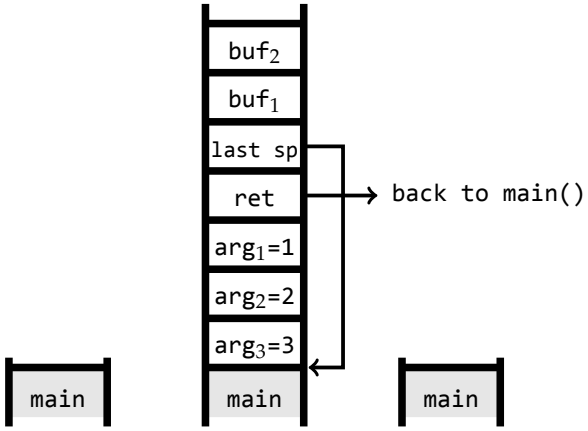
One such "benign" design decision is how the memory is laid out into different regions for each process.



The text region contains the program code (usually this region is read-only). The heap stores all data the programmer explicitly allocates. For us the most interesting region is the stack, which contains data mostly associated with the "control flow" of the program. Notice that the stack grows from a higher addresses to lower addresses. That means that older items on the stack will be stored behind newer items. Let's look a bit closer what happens with the stack. Consider the the trivial C program.

```
1 void foo(int a, int b, int c) {
2    char buffer1[6] = "abcde";
3    char buffer2[10] = "123456789";
4 }
5
6 void main() {
7   foo(1,2,3);
8 }
```

The main function calls `foo` with three argument. Foo contains two (local) buffers. The interesting point is what will the stack looks like after Line 3 has been executed? The answer is as follows:



On the left is the stack before `foo` is called; on the right is the stack after `foo` finishes. The function call to `foo` in Line 7 pushes the arguments onto the stack in reverse order—shown in the middle. Therefore first 3 then 2 and finally 1. Then it pushes the return address to the stack where execution should resume once `foo` has finished. The last stack pointer (`sp`) is needed in order to clean up the stack to the last level—in fact there is no cleaning involved, but just the top of the stack will be set back. The two buffers are also on the stack, because they are local data within `foo`.

Another part of the "conspiracy" is that library functions in C look typically as follows:

```
void strcpy(char *src, char *dst) {
  int i = 0;
  while (src[i] != "\0") {
    dst[i] = src[i];
    i = i + 1;
  }
}
```

This function copies data from a source `src` to a destination `dst`. The important point is that it copies the data until it reaches a zero-byte (`"\0"`).

A Crash-Course on GDB

- `(l)ist n` – listing the source file from line `n`
- `disassemble fun-name`
- `run args` – starts the program, potential arguments can be given
- `(b)reak line-number` – set break point
- `(c)ontinue` – continue execution until next breakpoint in a line number
- `x/nxw addr` – print out `n` words starting from address `addr`, the address could be `$esp` for looking at the content of the stack
- `x/nxb addr` – print out `n` bytes

If you want to know more about buffer overflow attacks, the original Phrack article "Smashing The Stack For Fun And Profit" by Elias Levy (also known as Aleph One) is an engaging read:

`http://phrack.org/issues/49/14.html`

This is an article from 1996 and some parts are not up-to-date anymore. The article called "Smashing the Stack in 2010"

`http://www.mgraziano.info/docs/stsi2010.pdf`

updates, as the name says, most information to 2010.

```c
#include <string.h>
#include <stdio.h>
#include <stdlib.h>

// Since gets() is insecure and produces lots
// of warnings, thereofre I use my own input
// function instead.
int i;
char ch;

void get_line(char *dst) {
  char buffer[8];
  i = 0;
  while ((ch = getchar()) != '\n') {
    buffer[i++] = ch;
  }
  buffer[i] = '\0';
  strcpy(dst, buffer);
}

int match(char *s1, char *s2) {
  while(*s1 != '\0' && *s2 != '\0' && *s1 == *s2){
    s1++; s2++;
  }
  return( *s1 - *s2 );
}

void welcome() { printf("Welcome!\n"); exit(0); }
void goodbye() { printf("Wrong identity, exiting!\n"); exit(1); }

int main(){
  char name[8];
  char pw[8];

  printf("login: ");
  get_line(name);
  printf("password: ");
  get_line(pw);

  if(match(name, pw) == 0)
    welcome();
  else
    goodbye();
}
```

Figure 1: A suspicious login implementation.