

```

int password_verify(){
    // Assume password is of length 6
    char password_stored[7]; // one extra for \0
    char received_password[7];
    FILE *fp;

    // Program reads password from file
    fp = fopen("secret_file", "r");
    fscanf(fp, "%s", password_stored);
    fclose(fp);

    // Program receives user-input
    printf("Enter 6 letter password: ");
    scanf("%s", received_password);

    // Verify password char-by-char
    int i;
    for(i=0; i<6; i++){
        if(received_password[i] != password_stored[i]){
            printf("Password not matched\n");
            exit(-1);
        }
    }
    printf("Password matched! Welcome!\n");
    secret_function();
    return 0;
}

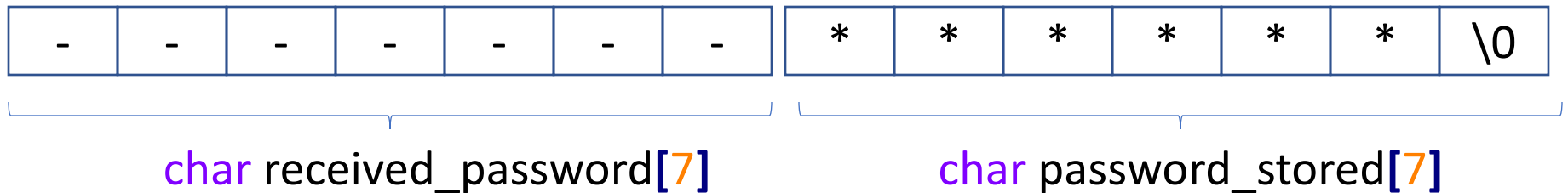
```

Here is an example of password verification.

If user-provided password matches with the stored password, then a secret function is called.

```
int password_verify(){  
    // Assume password is of length 6  
    char password_stored[7]; // one extra for \0  
    char received_password[7];  
    FILE *fp;  
    ...  
}
```

Memory allocation of two arrays in the Stack frame of password_verify()



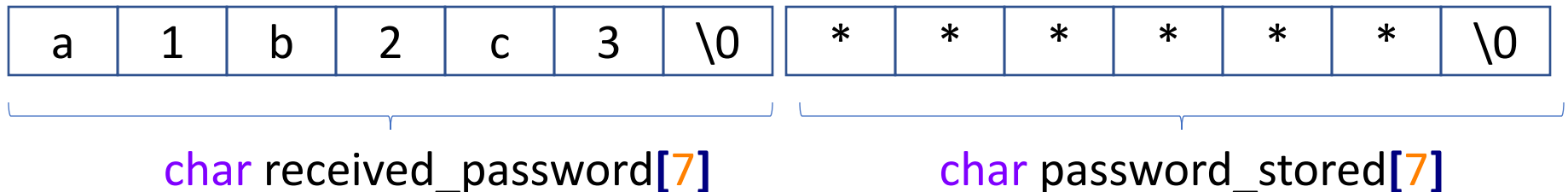
* is used to represent secret char

```

int password_verify(){
    // Assume password is of length 6
    char password_stored[7]; // one extra for \0
    char received_password[7];
    FILE *fp;
    ...
    // Program receives user-input
    printf("Enter 6 letter password: ");
    scanf("%s", received_password);
    ...
}

```

User provides a 6-char long string, say “a1b2c3” as a password



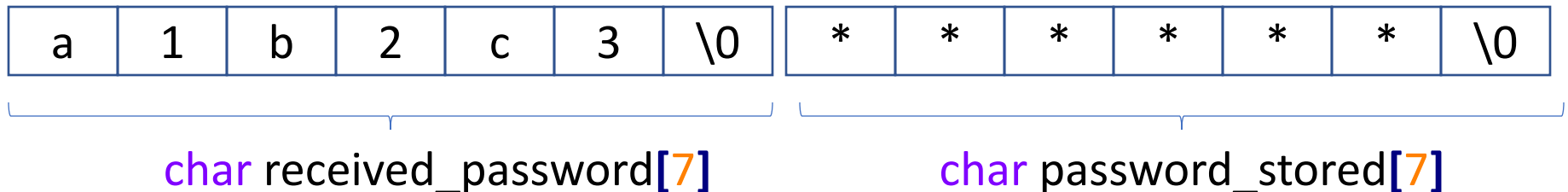
* is used to represent secret `char`

```

int password_verify(){
    // Assume password is of length 6
    char password_stored[7]; // one extra for \0
    char received_password[7];
    FILE *fp;
    ...
    // Program receives user-input
    printf("Enter 6 letter password: ");
    scanf("%s", received_password);
    ...
}

```

User provides a 6-char long string, say “a1b2c3” as a password



* is used to represent secret `char`

Password verification fails as soon as a mismatch is found

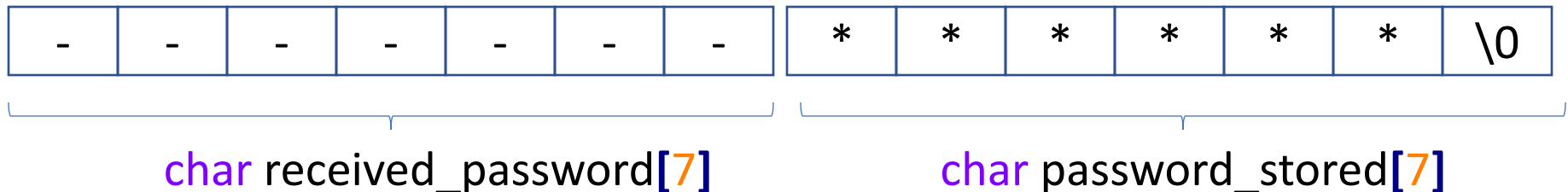
```

int password_verify(){
    // Assume password is of length 6
    char password_stored[7]; // one extra for \0
    char received_password[7];
    FILE *fp;
    ...
    // Program receives user-input
    printf("Enter 6 letter password: ");
    scanf("%s", received_password);
    ...
}

```

‘Nasty’ user enters a much longer string, say
“aaaaaaaaaaaaaaaaaaaaa”

What happens next?



* is used to represent secret `char`

```

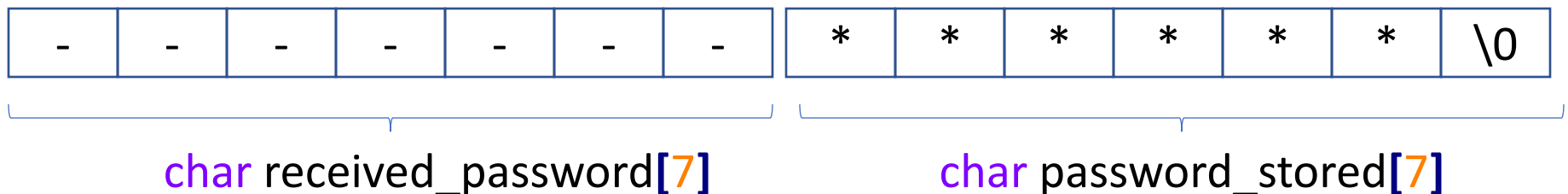
int password_verify(){
    // Assume password is of length 6
    char password_stored[7]; // one extra for \0
    char received_password[7];
    FILE *fp;
    ...
    // Program receives user-input
    printf("Enter 6 letter password: ");
    scanf("%s", received_password);
    ...
}

```

scanf() doesn't check length of input string. It writes everything into the buffer.

'Nasty' user enters a much longer string, say
 "aaaaaaaaaaaaaaaaaaaaa"

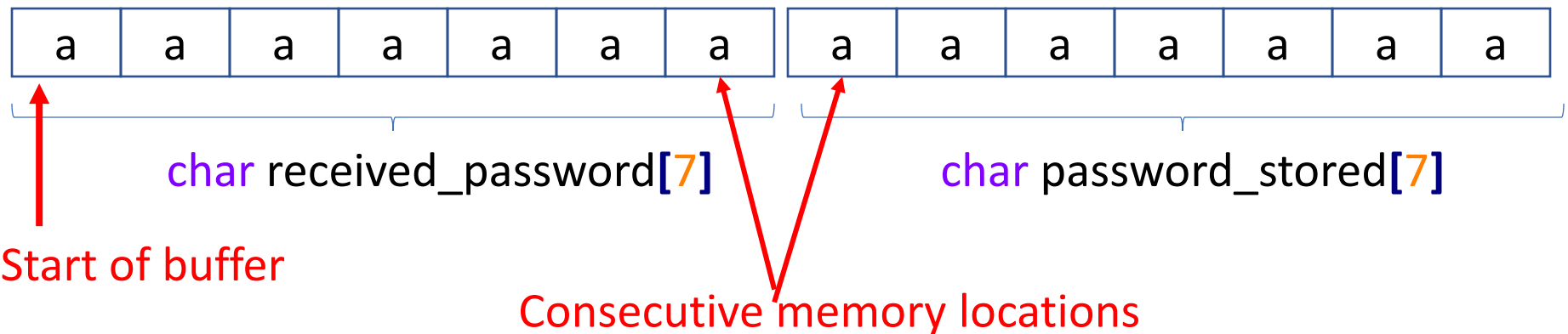
What happens next?



* is used to represent secret char

scanf() doesn't check
length of input string.
It writes everything into
the buffer.

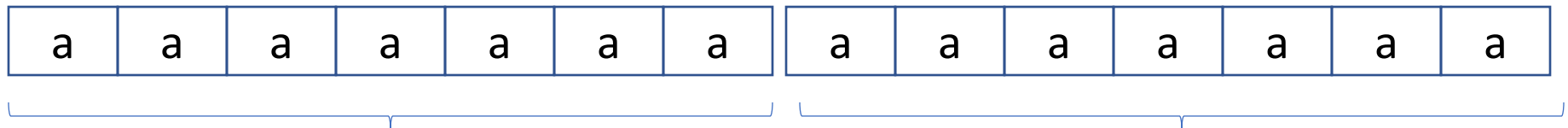
Correct password is overwritten



```

int password_verify(){
    ...
    // Verify password char-by-char
    for(i=0; i<6; i++){
        if(received_password[i] != password_stored[i]){
            printf("Password not matched\n");
            exit(-1);
        }
    }
    printf("Password matched! Welcome!\n");
    secret_function();
    ...
}

```



char received_password[7] ↔ char password_stored[7]

Now, they match

Attacker wins in calling secret_function();



WIKIPEDIA
The Free Encyclopedia

[Main page](#)
[Contents](#)
[Featured content](#)
[Current events](#)
[Random article](#)
[Donate to Wikipedia](#)
[Wikipedia store](#)

[Not logged in](#) [Talk](#) [Contributions](#) [Create account](#) [Log in](#)

[Article](#)

[Talk](#)

[Read](#)

[Edit](#)

[View history](#)



Buffer overflow

From Wikipedia, the free encyclopedia

In [information security](#) and [programming](#), a **buffer overflow**, or **buffer overrun**, is an [anomaly](#) where a [program](#), while writing [data](#) to a [buffer](#), overruns the buffer's boundary and [overwrites](#) adjacent [memory](#) locations.

Buffers are areas of memory set aside to hold data, often while moving it from one section of a program to another, or between programs. Buffer overflows can often be triggered by malformed inputs; if one assumes all inputs will be smaller than a certain size and the buffer is created to be that size, then an anomalous transaction that produces more data could cause it to write past the end of the buffer. If this overwrites adjacent data or executable code, this may result in erratic program behavior, including memory access errors, incorrect results, and [crashes](#).

This bug is known as the 'Buffer overflow bug'

How to prevent buffer overflow

```
int password_verify(){  
    ...  
    // Program receives user-input  
    printf("Enter 6 letter password: ");  
    //scanf("%s", received_password);  
    fgets(received_password, sizeof(received_password), stdin);  
    ...  
}
```

This is how you read a string and prevent buffer overflows in C:

```
fgets(buffer, sizeof(buffer), stdin);
```

`fgetc()` ensures that only `sizeof(buffer)`-bytes are read from standard input.