Insecure coding in C

Olve Maudal

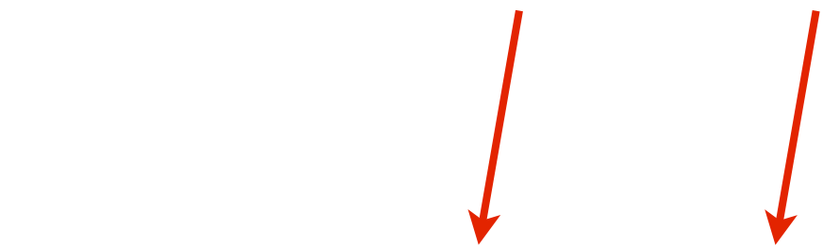
Let's turn the table. Suppose your goal is to deliberately create buggy programs in C with serious security vulnerabilities that can be easily exploited. In this session we will show some cool programming techniques that hides vulnerable code just waiting to be exploited now or later.

20 minute presentation, Cisco SecConX London, May 7 2014


jackass

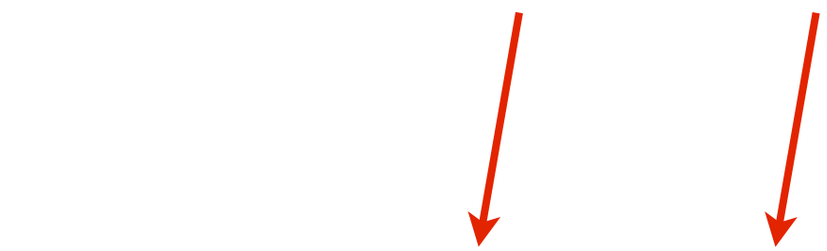
```
c = a() + b();
```

```
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;
```

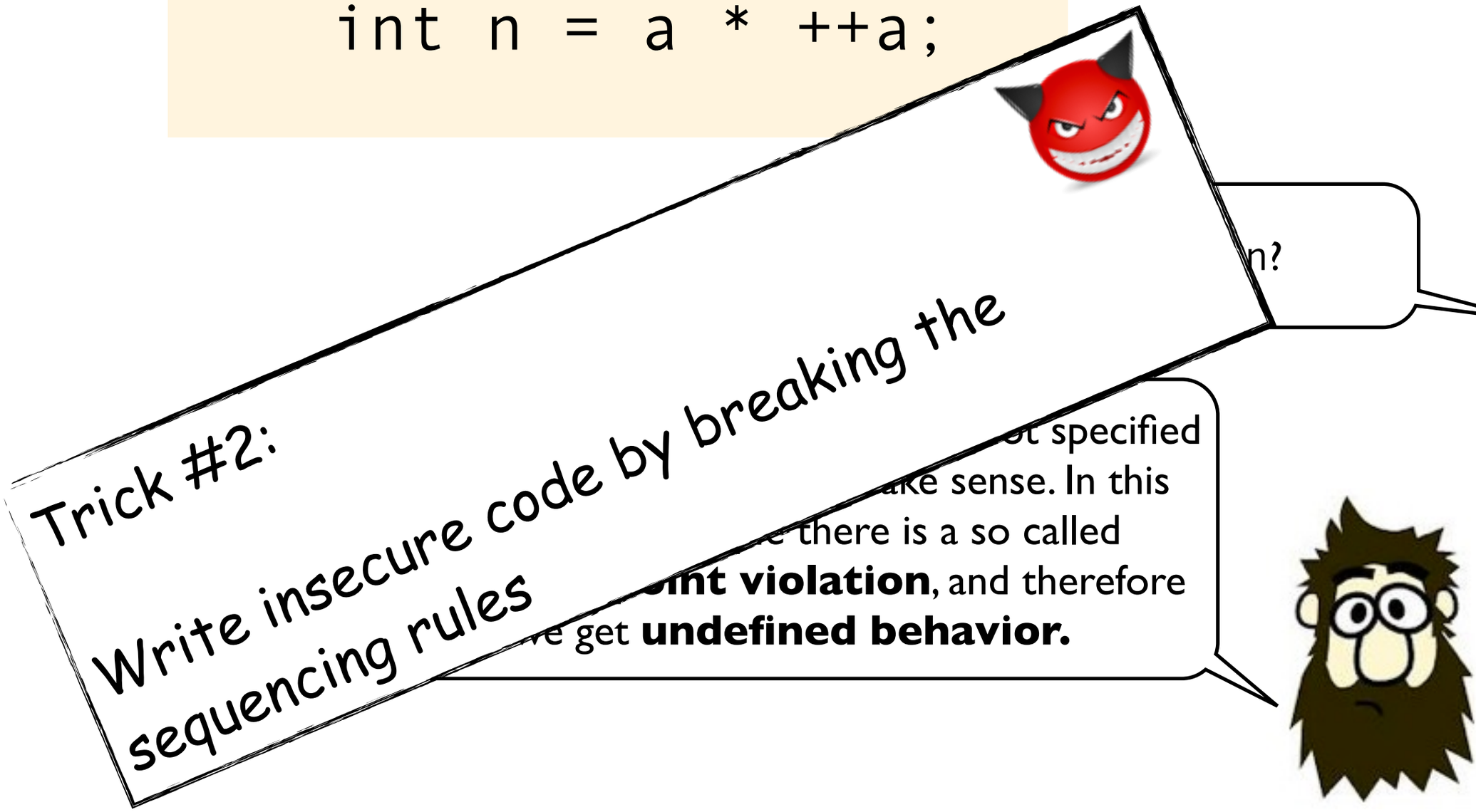
What is the value of n?

```
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;
```



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
```

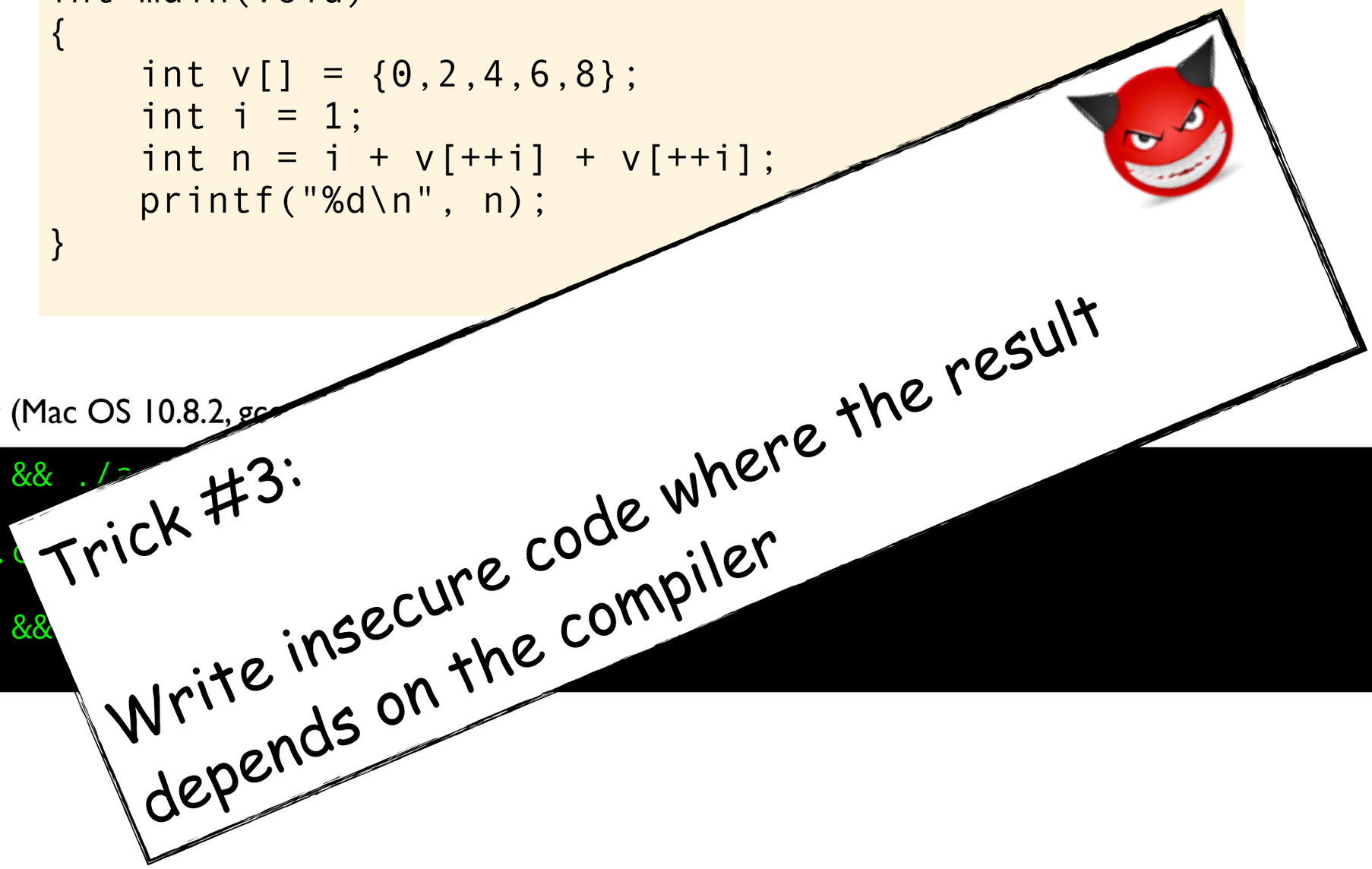
What do you think will **actually** happen if you compile, link and run it in your development environment?

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
```



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.

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.

(*) see http://www.pvv.org/~oma/UnspecifiedAndUndefined_ACCU_Apr2013.pdf for detailed explanation of this phenomenon

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);
}
```

Trick #4:

Write insecure code by knowing the blind spots of your compilers

On my computer (Mac OS 10.8.2, gcc

```
$ gcc -std=c99 -O
12
$ clang -std=c
11
$ icc -std=c99
13
```

a.out

&& ./a.out

The point is that the C standard does **not** require compilers to diagnose "illegal" code.

(*) see http://www.pvv.org/~oma/UnspecifiedAndUndefined_ACCU_Apr2013.pdf for detailed explanation of this phenomenon

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

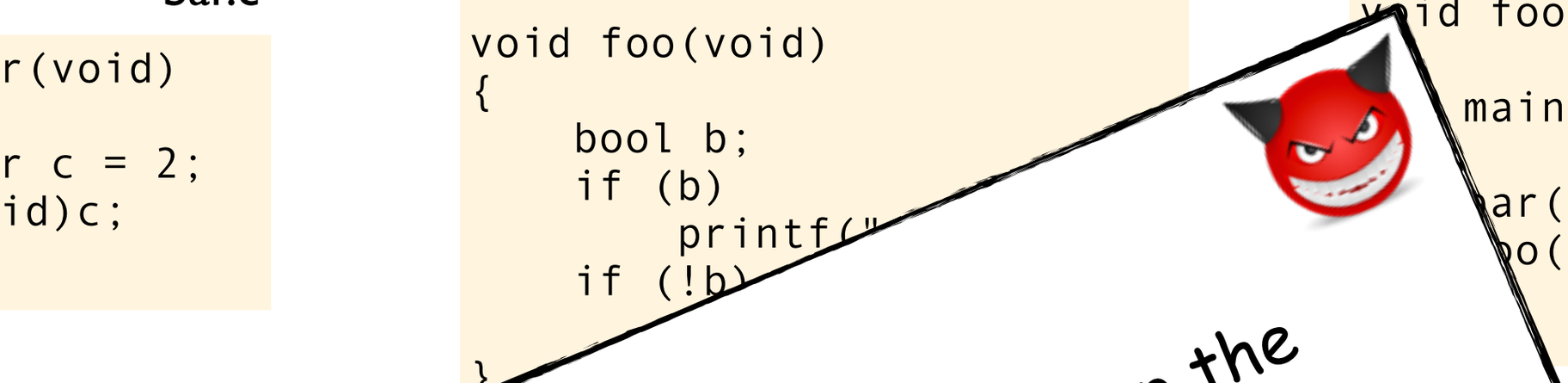
```
void bar(void)
{
    char c = 2;
    (void)c;
}
```

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
```

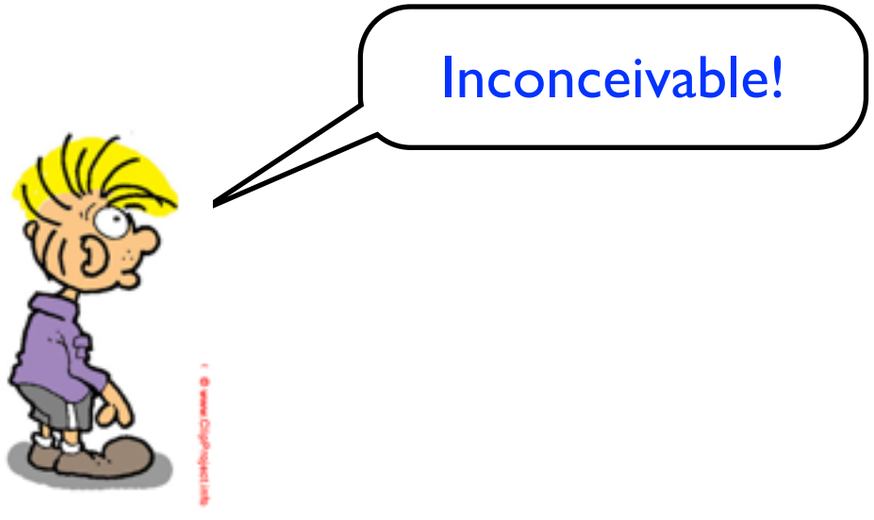
deep_thought.c

```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
```



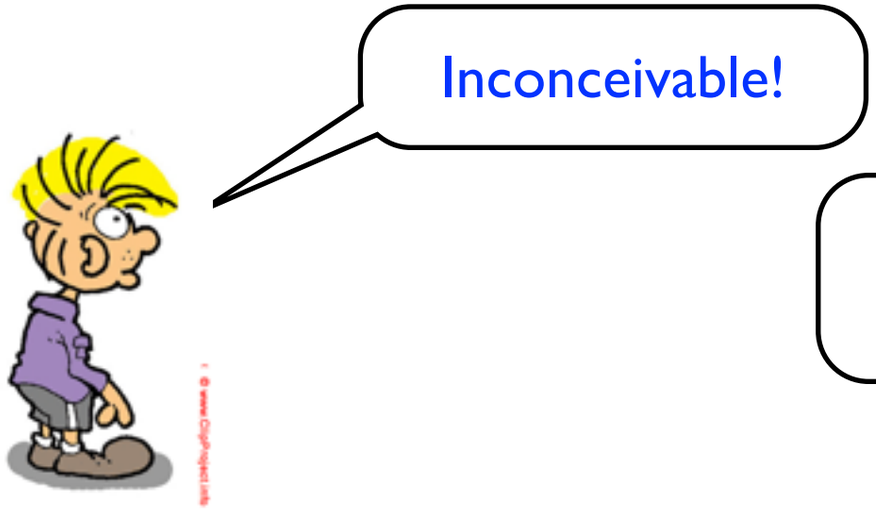
deep_thought.c

```
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
```

```
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...
}
```

```
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
```

```
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
```

```
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
```

```
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
```

```
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
```

```
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
```

```
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
```

```
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
```

```
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
```

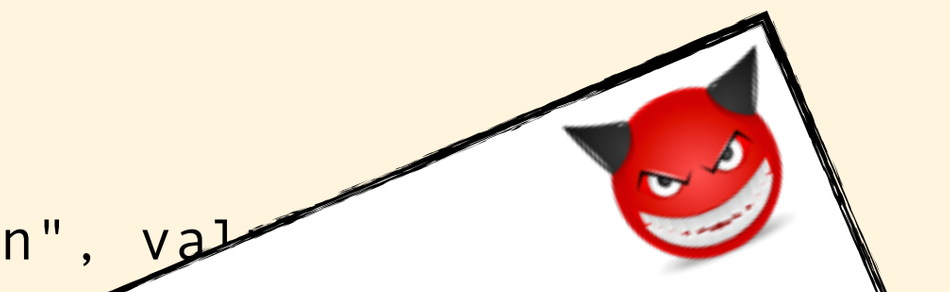
```
void poke(unsigned char * ptr, size_t offset,
          unsigned char * end, unsigned char value)
{
    printf("ptr=%p offset=%.8zx end=%p", ptr, offset, end);
    if (ptr + offset >= end) {
        printf(" --> out of bounds\n");
        return;
    }
    if (ptr + offset < ptr) {
        printf(" --> 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
```

```
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", val
    // TODO: implement this...
}
```



Trick #7:

Write insecure code by letting the optimizer remove apparently critical code

```
$ cc -m32 -O2 poke.c poke m
ptr=0xfffffffa offset=0
ptr=0xfffffffa off
ptr=0xfffffffa
ptr=0xffff
ptr=0xfffff
ptr=0xffffff
ptr=0xfffffff
ptr=0xfffffffa
ptr=0xfffffffa
ptr=0xfffffffa
```

```
                                                  0xfffffffa
                                                  into 0xfffffffb
                                                  ke 42 into 0xfffffffc
                                                  > out of bounds
                                                  d --> out of bounds
                                                  ffffffd --> out of bounds
                                                  =0xfffffffd --> poke 42 into 0x0
                                  end=0xfffffffd --> poke 42 into 0x1
                                  end=0xfffffffd --> poke 42 into 0x2
                                  end=0xfffffffd --> poke 42 into 0x3
```

```
#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");
}
```

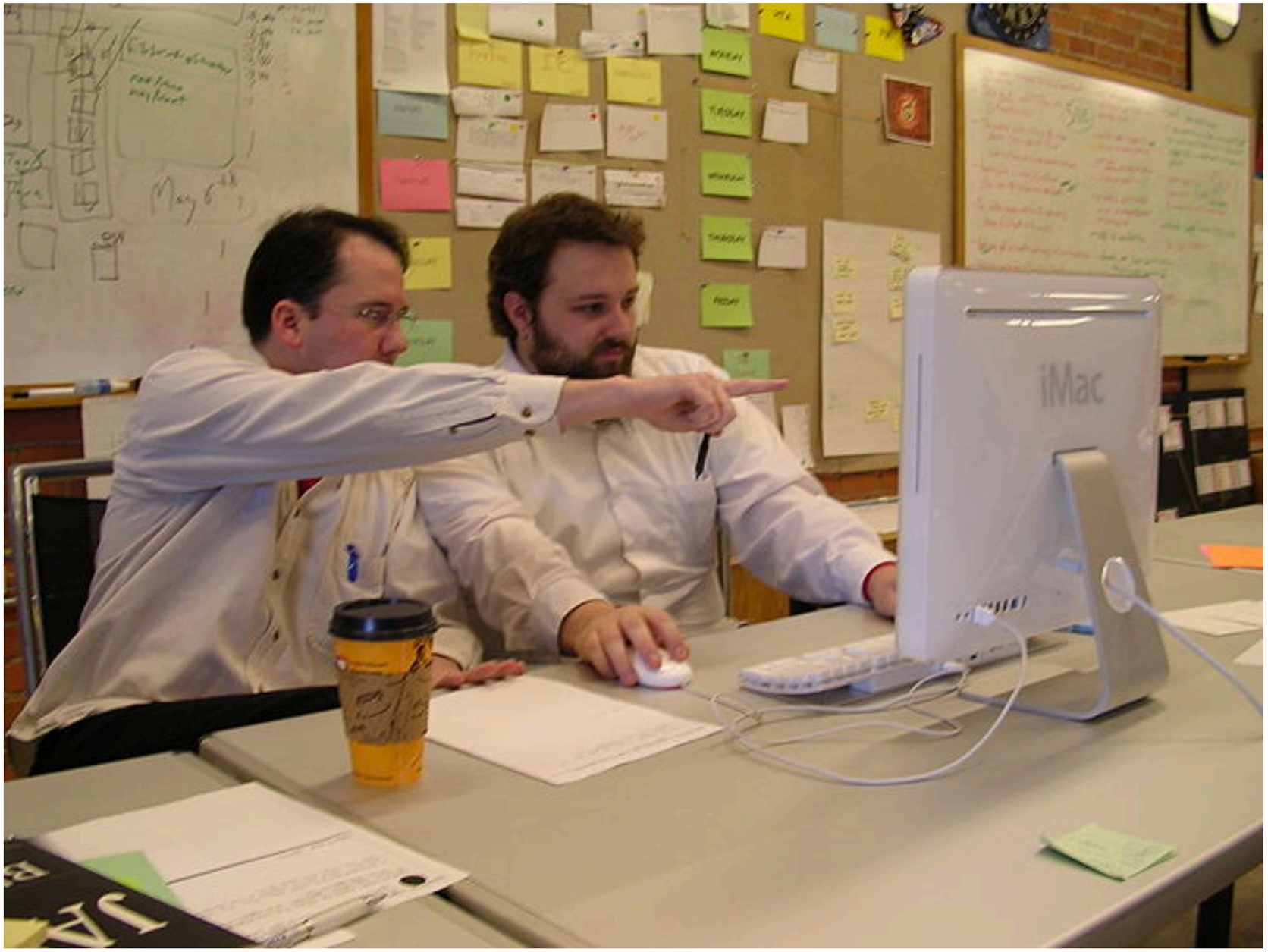
```
foo.c && ./a.out
David
globalthermonuclJoshua
```

```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

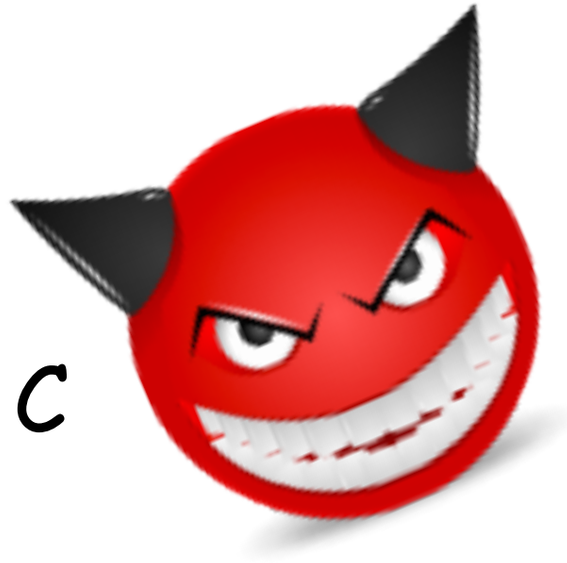
```
foo.c && ./a.out
David
globalthermonuclJoshua
```


iMac


Trick #0:
Never ever let other programmers review your code

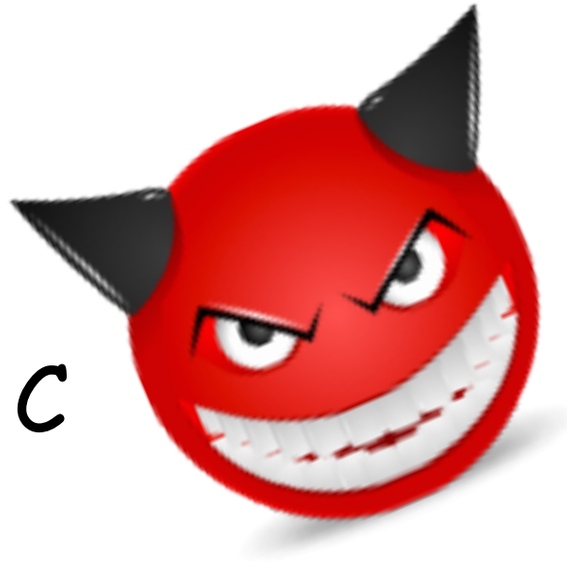
Some tricks for writing insecure code in C

Some tricks for writing insecure code in C



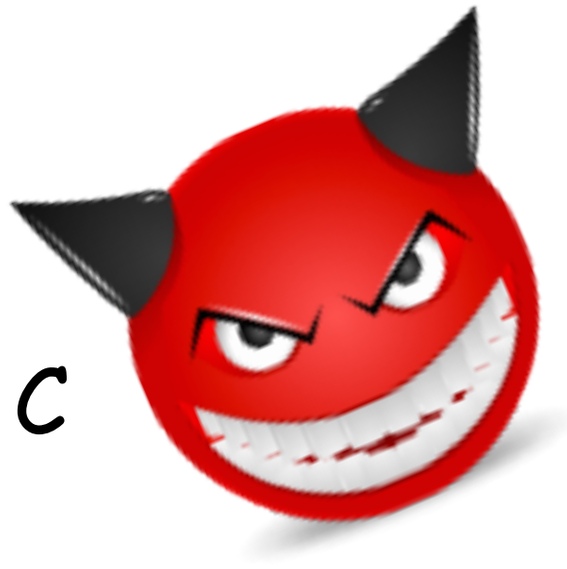
#1 Write insecure code by depending on a particular evaluation order

Some tricks for writing insecure code in C



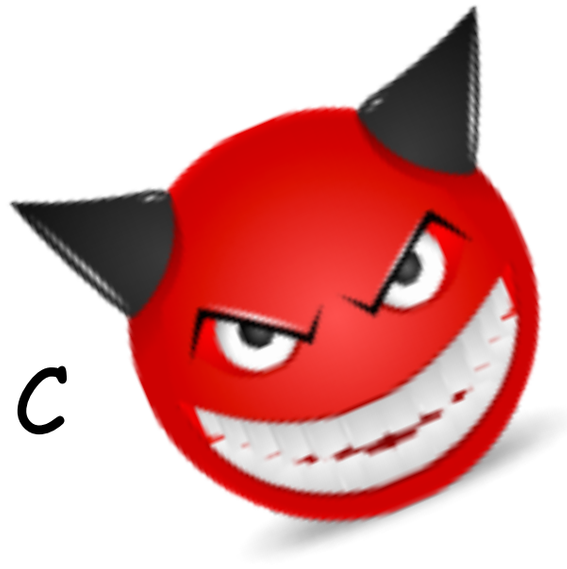
#1 Write insecure code by depending on a particular evaluation order
#2 Write insecure code by breaking the sequencing rules

Some tricks for writing insecure code in C



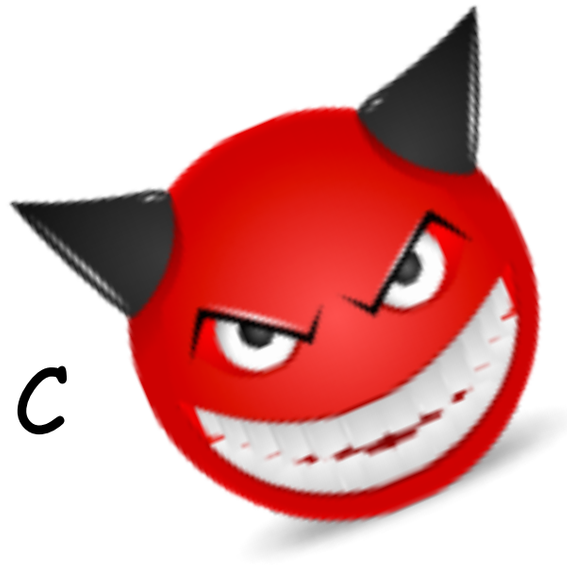
#1 Write insecure code by depending on a particular evaluation order
#2 Write insecure code by breaking the sequencing rules
#3 Write insecure code where the result depends on the compiler

Some tricks for writing insecure code in C



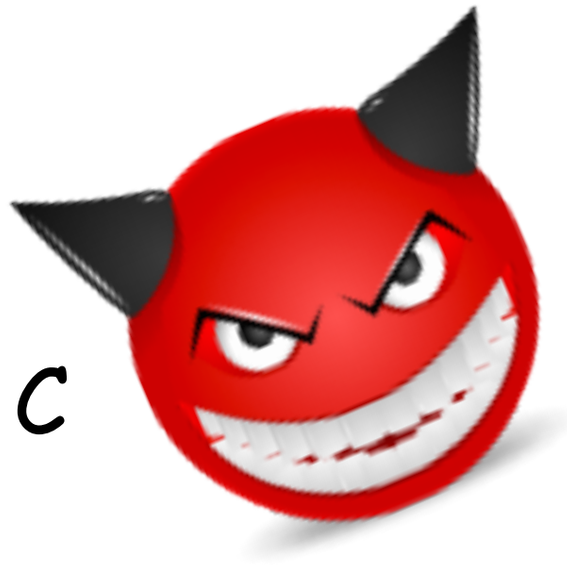
#1 Write insecure code by depending on a particular evaluation order
#2 Write insecure code by breaking the sequencing rules
#3 Write insecure code where the result depends on the compiler
#4 Write insecure code by knowing the blind spots of your compilers

Some tricks for writing insecure code in C



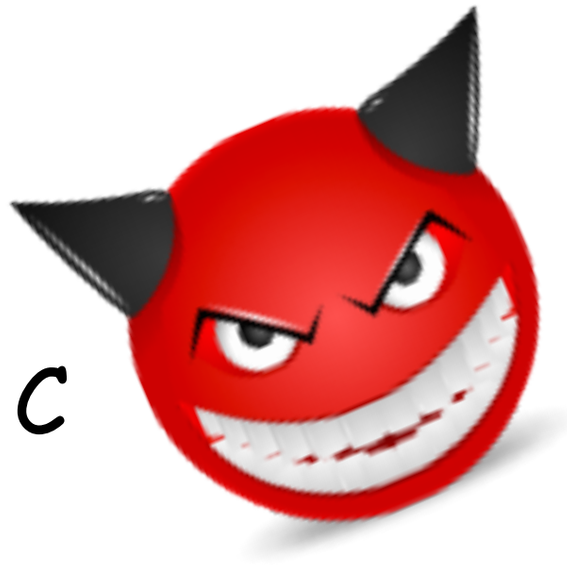
#1 Write insecure code by depending on a particular evaluation order
#2 Write insecure code by breaking the sequencing rules
#3 Write insecure code where the result depends on the compiler
#4 Write insecure code by knowing the blind spots of your compilers
#5 Write insecure code by messing up the internal state of the program.

Some tricks for writing insecure code in C



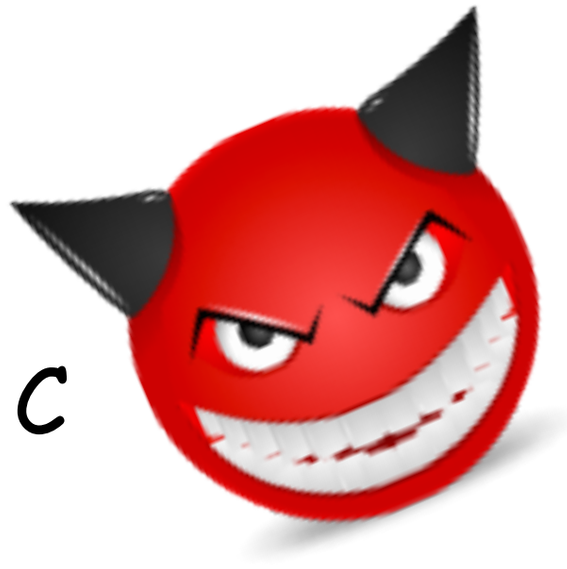
#1 Write insecure code by depending on a particular evaluation order
#2 Write insecure code by breaking the sequencing rules
#3 Write insecure code where the result depends on the compiler
#4 Write insecure code by knowing the blind spots of your compilers
#5 Write insecure code by messing up the internal state of the program.
#6 Write insecure code by only assuming valid input values

Some tricks for writing insecure code in C



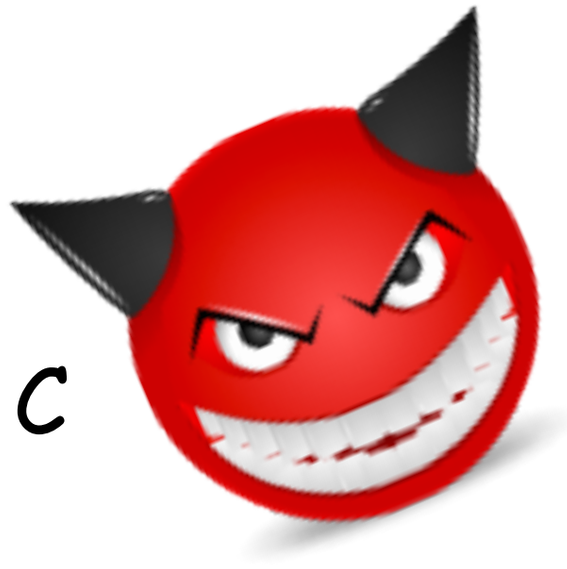
#1 Write insecure code by depending on a particular evaluation order
#2 Write insecure code by breaking the sequencing rules
#3 Write insecure code where the result depends on the compiler
#4 Write insecure code by knowing the blind spots of your compilers
#5 Write insecure code by messing up the internal 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 writing insecure code in 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

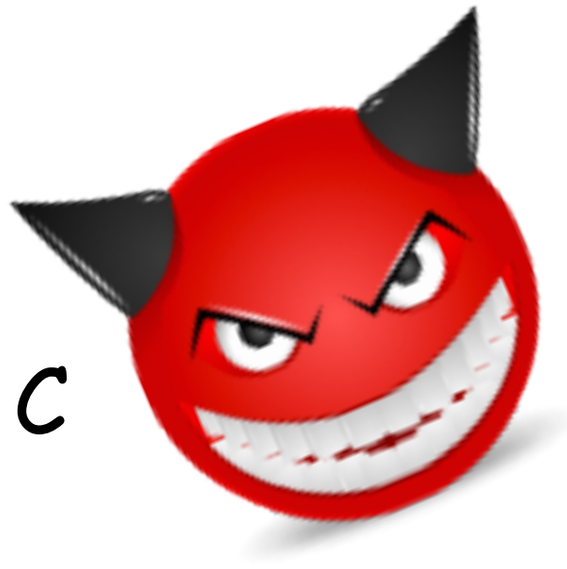
Some tricks for writing insecure code in C



#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

... and of course, there are plenty more tricks not covered here...

Some tricks for writing insecure code in 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

... and of course, there are plenty more tricks not covered here...

!