Input validation
- wrap-up discussion &
complementary aspects -

Questões de Segurança em Engenharia de Software (QSES)
Mestrado em Segurança Informática
Departamento de Ciência de Computadores
Faculdade de Ciências da Universidade do Porto

Eduardo R. B. Marques, edrdo@dcc.fc.up.pt

General context

- **“All input is evil!” or “Trust no input!” are common security “mantras”.**

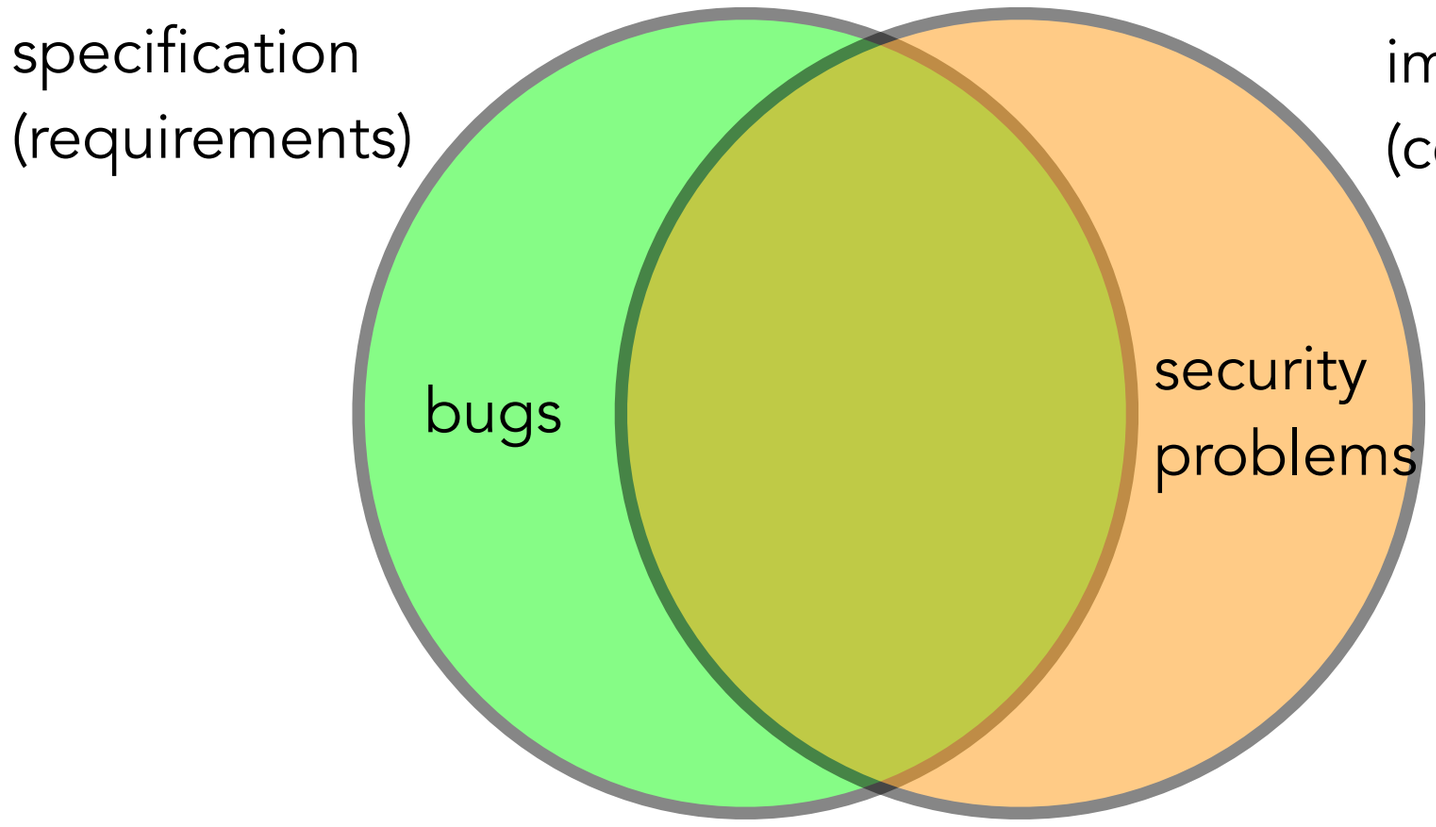
- Over these last classes, we learned why …

- Wrap-up discussion:
  - What is an “input” ? We’ve seen several kinds …
  - The notion of trust boundary.
  - Syntactic vs semantic input validation
  - A brief look at a few particular techniques and input validation issues (with an eye on some themes we will cover in later classes)

What is an “input” ?

- Input:
  - every item of data that comes from an external source and affects program behavior
- Possible data sources
  - Command line arguments
  - Configuration data (files, environment vars, etc)
  - Network servers
  - Database
  - File system
  - Shared memory
  - Hardware devices
  - …

Evil input >> “unintended functionality”



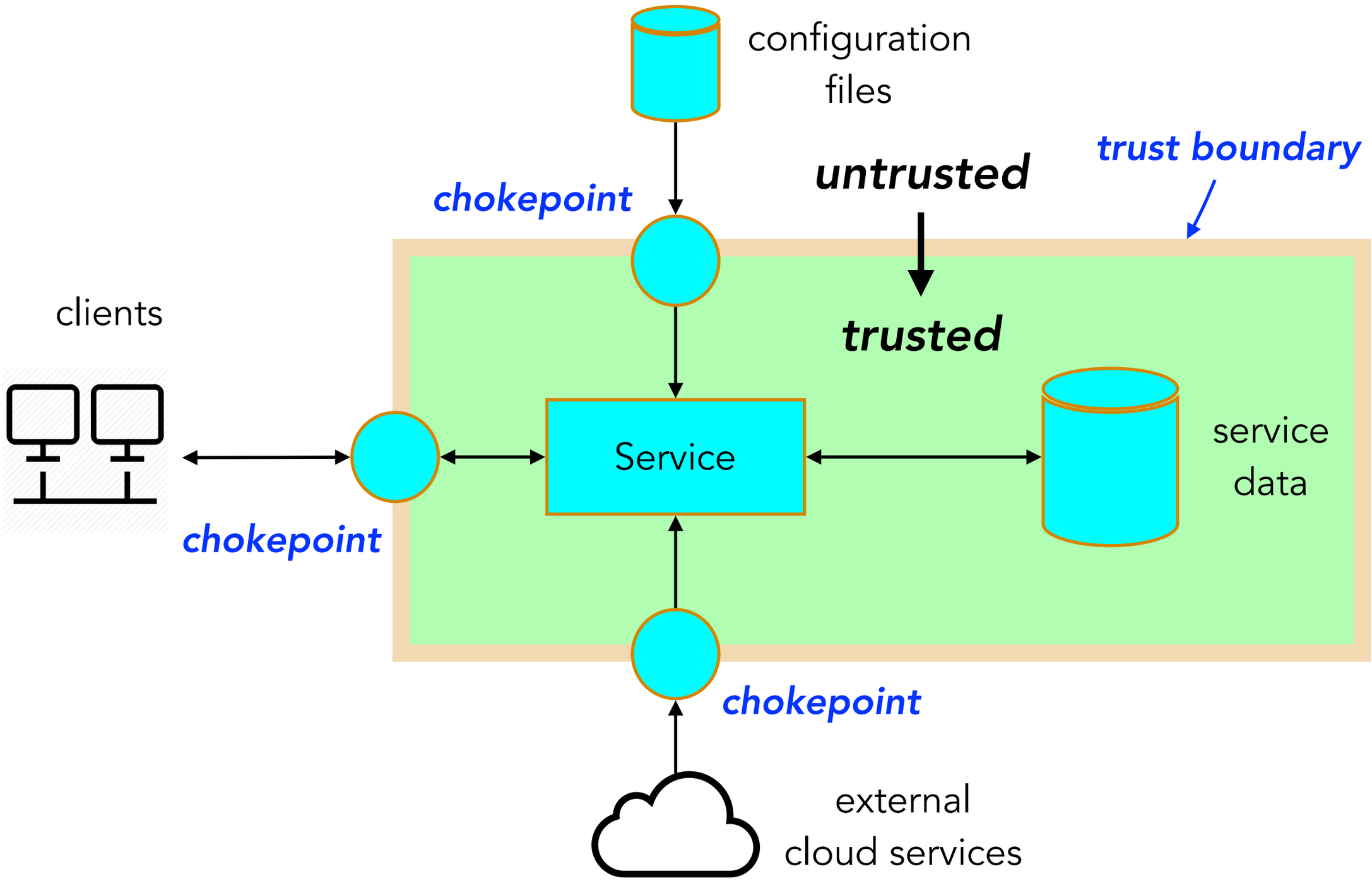
[from class 1]

- **A program executes according to the input it takes !**
- Unintended functionality / security vulnerabilities arise when untrusted input is let on the loose …
- Recall quote from class 1: **“Reliable software does what it is supposed to do. Secure software does what is supposed to do, and nothing else.”**

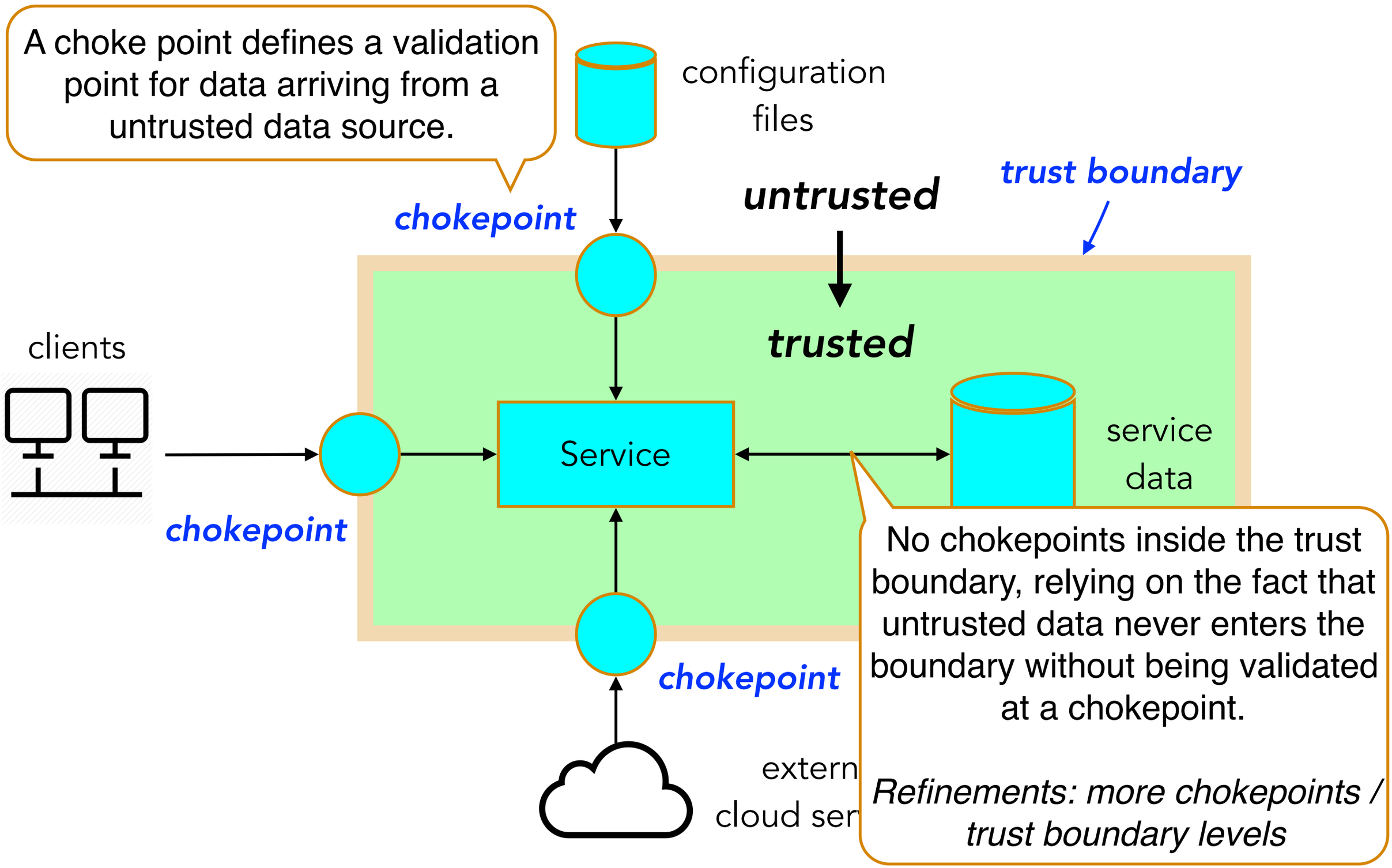
“All input is evil” / “Trust no input”

- **The input sources define the attack surface.**
  - Thus “All input is evil” (or “trust no input”) is a common mantra in secure software development.
- What may be wrong with the input?
  - invalid format => we need **syntactic** checks …
  - be well-formed but convey inappropriate data => we need **semantic** checks …
- Input validation mechanisms should sit at **trust boundaries:**
  - points in a program where the level of trust of a data item is changed (promoted or demoted).
  - **Chokepoint**: validation point at trust boundary transition
- Evil input should not reach **security-sensitive data sinks**:
  - **Sink**: point in a program that consumes data (database access, OS command execution, …)

Trust boundary [Howard & Leblanc, chap. 10]



Trust boundary [Howard & Leblanc, chap. 10]



Trust boundary violation

- Trust boundary violation (OWASP CWE-501)
  - “A trust boundary can be thought of as line drawn through a program. On one side of the line, data is untrusted. On the other side of the line, data is assumed to be trustworthy.”
  - “**A trust boundary violation occurs when a program blurs the line between what is trusted and what is untrusted**. By combining trusted and untrusted data in the same data structure, it becomes easier for programmers to mistakenly trust unvalidated data.”

Trust boundary violation (2)

Example 1

The following code accepts an HTTP request and stores the username parameter in the HTTP session object before checking to ensure that the user has been authenticated.

Example Language: **Java** *(Bad Code)*

```
usrname = request.getParameter("usrname");
if (session.getAttribute(ATTR_USR) == null) {
  session.setAttribute(ATTR_USR, usrname);
}
```

Example Language: **C#** *(Bad Code)*

```
usrname = request.Item("usrname");
if (session.Item(ATTR_USR) == null) {
  session.Add(ATTR_USR, usrname);
}
```

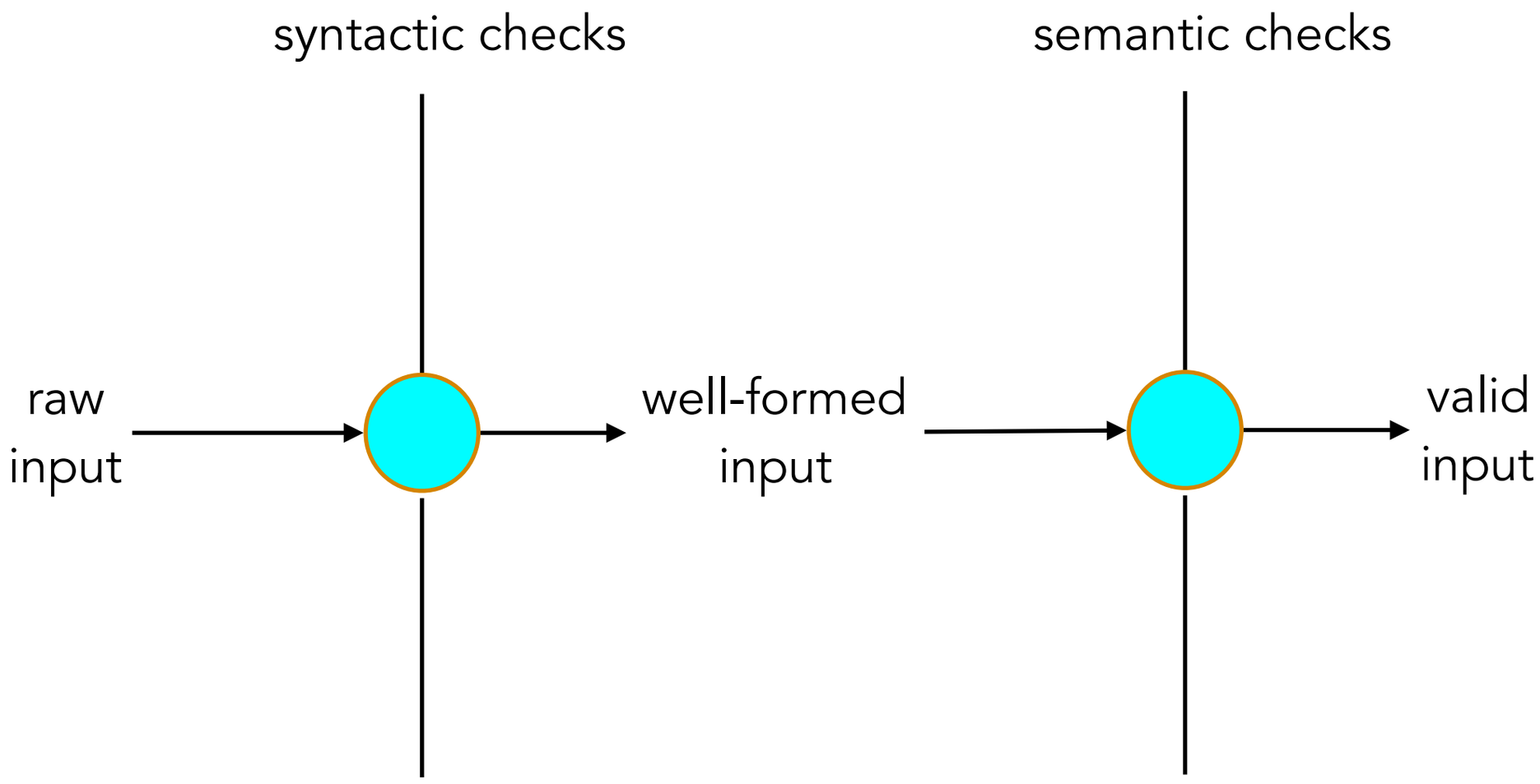
Without well-established and maintained trust boundaries, programmers will inevitably lose track of which pieces of data have been validated and which have not. This confusion will eventually allow some data to be used without first being validated.

- Example from Trust boundary violation (OWASP CWE-501)

Syntactic and semantic checks

- What may be wrong with the input?
  - invalid format => we need syntactic chokepoints …
  - be well-formed but convey inappropriate data => we need semantic chokepoints …
- **Syntactic chokepoints**
  - are placed at points where input data is read
  - validate (and possibly sanitize) the raw format of input without other concerns
- **Semantic chokepoints**
  - are placed at points in the application logic is accessed
  - possibly deal with inputs from (or affected by) disparate sources, previously validated by syntactic chokepoints
  - validate that the conveyed in appropriate in semantic terms taking the application logic into account
  - usually perform no sanitization, should fail gracefully!

Different checks / different trust boundaries



- Defence in depth principle at work
  - Syntactic chokepoints → right after reading data
  - Semantic chokepoints → part of the application logic

A few mechanisms for syntactic checks

- We have seen several examples comprising the use of sanitisation for input validation or output encoding. We have also mentioned white-lists and black-lists, but discuss them now in more detail.
- Whitelist
  - defines a set comprising all inputs that are valid
  - all others will be rejected
- Blacklist
  - defines a set of all inputs that should be rejected
  - all others will be accepted
- How to implement black-lists / white-lists
  - Regular expressions can be useful in many cases, we will go through a simple examples
  - more sophisticated input may be dealt with by special-purpose parsers e.g. XML and JSON schema-based parsers
    - XML schema example at W3 schools
    - JSON schema

Whitelist — simple example

```
private static final String[] ALLOWED_FILE_EXTENSIONS = {
    ".gif", ".jpeg", ".png"
};

static boolean isValidFileExtension(String fileName) {
   for (String ext : ALLOWED_FILE_EXTENSIONS) {
      if (fileName.endsWith(ext)) {
        return true;
      }
   }
   return false;
}
```

- Whitelist defines the set of valid inputs. All others will be rejected.

Blacklist — simple example

```
private static final String[] DISALLOWED_FILE_EXTENSIONS = {
     ".exe", ".com", ".bat"
};

static boolean isValidFileExtension(String fileName) {
    for (String ext : DISALLOWED_FILE_EXTENSIONS) {
      if (fileName.endsWith(ext)) {
        return false;
      }
    }
    return true;
}
```

- Blacklist defines the set of invalid inputs. All others will be accepted.
- Whitelists vs blacklists: what do you think it's best?

Blacklist using regular expressions

```
private static final Pattern PHONE_NUMBER_PATTERN
    = Pattern.compile("\\+?[0-9\\- ]+");

static boolean isValidPhoneNumber(String s) {
  return PHONE_NUMBER_PATTERN.matcher(s).matches();
}
```

- Simplistic blacklist-style check: allow optional ‘+’ at the start, constrain rest of the string to contain only numbers, spaces, or ‘-‘.
- "+ - - - - “ is considered valid.

Whitelist based on regular expressions

```
// Regular expression for dates in a YYYY-MM-DD format
  private static final Pattern DATE_PATTERN
      = Pattern.compile("(\\d{4})-(\\d{2})-(\\d{2})");

  static boolean isValidDate(String s) {
    Matcher m = DATE_PATTERN.matcher(s);
    if (! m.matches() )
      return false;

    int year = Integer.parseInt(m.group(1));
    int month = Integer.parseInt(m.group(2));
    int day = Integer.parseInt(m.group(3));

    return month >= 1 &&
           month <= 12 &&
           day >= 1 &&
           day <= daysInMonth(month, year);
  }
```

- Dates are validated with a YYYY-MM-DD format.
- Semantic check also to check that date is valid.

Whitelisting vs. blacklisting

- Whitelists are generally preferable
  - they concretely define what good inputs are
  - but in some scenarios they can be too restrictive / unfeasible
- Blacklists only identifies a set of bad inputs.
  - The set may be incomplete or hard to enumerate … providing a false sense of security. Chances are that some bad inputs are not filtered out.
  - There is a vulnerably class for incomplete black-lists — CWE-184.
  - Blacklists may however be simpler to implement or more adequate in some cases, e.g., blacklists of domains associated with e-mail spamming.
- Further references (even beyond input validation)
  - Whitelisting vs blacklisting, OWASP “Input Validation Cheat Sheet”
  - Whitelisting vs. Blacklisting , Schneier on Security (short blog article by Bruce Schneier)

Semantic checks — the particular case of bounds checking

- Relatively easy to handle in most cases, but also easy to overlook/forget when programming …
- Check input length to avoid:
  - buffer overflows
  - unbounded input
  - …
- Check bounds of numeric values to avoid:
  - numeric overflows
  - type conversion issues
  - …

Checking for limits — buffer overflows & unbounded input

buffer overflows possible

```
// In C
char line[128];
gets(line);

// C++
char line[128];
cin >> line
```

unbounded input => CPU or memory DoS

```
// In C++
std::string line;
cin >> line

// Java
String str;
str = bufferedReader.readLine()
```

- Note:
  - We'll talk about buffer overflows in a future class

Checking for limits — bounds for numeric values & numeric overflows

Small Java fragment

```
    System.out.println(Integer.MIN_VALUE);   // -2147483648
    System.out.println(Integer.MAX_VALUE+1); // -2147483648
    int v = 1073741824; // 2^31
    System.out.println(2 * v); // -2147483648
    System.out.println(4 * v); // 0
    System.out.println(Integer.MIN_VALUE + Integer.MIN_VALUE); // 0
```

Now consider the logic below for a fictitious e-commerce site where input validation should be stronger. The value “to pay” will be 0 if price == 4 and quantity == 1073741824 (or vice-versa).

```
try(Scanner in = new Scanner(System.in)) {
      int price = in.nextInt();
      int quantity = in.nextInt();
      if (price <= 0 || quantity <= 0) {
        System.out.println("Invalid request!");
      } else {
        int toPay = price * quantity;
        System.out.printf("OK! The price to pay is %d\n", toPay);
      }
    }
```

Bounds checking — numeric values & type conversions

```
void* myAlloc(int sn) {
  size_t un = sn;
  printf("Allocating %lu bytes (sn=%d)\n", un, sn);
  return malloc(un);
}
```

int is a signed type
but size_t is unsigned!
if n < 0 , sn will be large,
possibly causing a vast amount
of memory to be allocated or
malloc to fail

```
caetano:qses edrdo$ ./myAlloc -1
Allocating 18446744073709551615 bytes (n=-1)
Allocated buffer: 0x0
caetano:qses edrdo$ ./myAlloc -10000
Allocating 18446744073709541616 bytes (n=-10000)
myAlloc(11705,0x7fffb551a3c0) malloc: *** mach_vm_map(size=18446744073709543424) failed
(error code=3)
*** error: can't allocate region
*** set a breakpoint in malloc_error_break to debug
Allocated buffer: 0x0
```