Secure Coding

Sicurezza (CT0539)   2023-24
Università Ca' Foscari Venezia

Riccardo Focardi
www.unive.it/data/persone/5590470
secgroup.dais.unive.it

Università
Ca'Foscari
Venezia

Motivation

Programming languages can be **unsafe**, especially when they allow for low-level access to memory

Languages such as C are particularly unsafe and require great attention from programmers but **any** programming language exhibits unsafe behaviours

We discuss how to write **safe** and **secure** programs in C

Standards

ISO/IEC TS 17961: establishes baseline requirements for **analyzers** and **compilers**

All requirements can be enforced by static analysis (**compile time**)

⇒ Discover **coding errors** without too many false positives

Has been applied in non-uniform, ad-hoc manners by different vendors

The SEI CERT C Coding Standard provides **rules** and **recommendation** from the security coding community

- **Rules** provide normative **requirements** for code
- **Recommendations** provide **guidance** to improve code **safety**, **reliability**, and **security**

⇒ Freely available!

Tools and incompleteness

Manual inspection of code is only possible for small programs

Static analysis tools are necessary for real-world applications

Properties that depend control-flow are in general **undecidable**, so static analysis tools cannot be 100% precise (cf. *halting* theorem)

False negative: failure to report a flaw

False positive: report nonexisting flaw

What is preferable?

False negatives should be avoided (insecure code). Tools try to ***err on the safe side*** giving false positives

⇒ however, too many false positives make programming hard!

Sound / Complete analysis

Sound: bad programs are all rejected, i.e., no false negatives (good programs might be rejected)

Complete: no good program is rejected, i.e., no false positives (bad programs might be accepted)

False Positives

False Negatives		No	Yes
	No	Sound and Complete	Sound with False Positives
	Yes	Complete with False Negatives	Unsound and incomplete

Goal: sound and complete for simple, syntactic rules. Otherwise, sound minimizing false positives

Taint analysis

Determines which values coming from program inputs can **influence** values used in a **risky** operation

Tainted source: Any source of external data that could be controlled by an attacker

Tainted value: Value derived or computed from a **tanited source** and has not been properly *sanitized*

Restricted sink: An argument of a function that is required to be in a *restricted* domain

Many library functions in C have restricted sinks

Example: strings are usually required to be NULL terminated. If not the function will **access subsequent memory**

Taint propagation

Taint is **propagated** through operations from operands to results unless the operation itself imposes **constraints** on the value of its result

Examples:

`strcpy(s1,s2)`: copies `s2` in `s1`

`strcat(s1,s2)`: appends `s2` to `s1`

⇒ if `s2` is tainted, also `s1` is tainted

Propagation can be **complex**: taint of one sort can propagate as taint of a different sort

Example 1: `strlen` if the string is not NULL-terminated

Example 2: An exit condition of a loop based on a tainted value taints all the values of variables modified in the loop

Taint propagation: example

```
char buffer[MAX],c;
int i,len;

memset(buffer,'\0',MAX);

// Reads chars from terminal
for (i=0; i<MAX &&( c = getchar())!=EOF; i++)
    buffer[i] = c;

// computes len for further loops
len = strlen(buffer);

// loops over len chars to process buffer
for (i=0; i< len ; i++) {
    // process the buffer ...
}
```

User input: tainted source

c is tainted

buffer is tainted (it is modified in the loop based on c)

len is tainted

variables modified in the loop are tainted

Taint propagation: example

An **off-by-one bug** in the first loop makes buffer non NULL-terminated in case of an input of MAX characters, which **propagates** over all tainted variables!

```
memset(buffer,'\0',MAX); // zeroes the buffer
// Reads chars from terminal (should stop at MAX-1!)
for (i=0; i<MAX & ( c = getchar() ) != EOF; i++)
    buffer[i] = c;
```

Example with MAX = 16:

```
$ echo -n "AAAAAAAAAAAAAAA" | ./taint_example
len=15, buffer=AAAAAAAAAAAAAAA

$ echo -n "AAAAAAAAAAAAAAAA" | ./taint_example
len=22, buffer=AAAAAAAAAAAAAAAAP8??tU
```

16 A's fill the buffer, the string becomes **non terminated** and 6 more chars are read! **len is 22** which is bigger than MAX-1!

Sanitization

Taint can be removed by **sanitization**

Two approaches:

Replacement: out of domain values for restricted sinks are **replaced** by in-domain values

Termination: out of domain value is detected and program either **terminates** or **skip** the code using that value

Example (replacement): we NULL-terminate the string

```
buffer[MAX-1] = '\0';
```

Example (termination): we check that it is null terminated

```
if (buffer[MAX-1] != '\0')
    exit(1);
```

⇒ `buffer` is now OK in restricted sinks requiring NULL-terminated strings

Secure Coding: SEI CERT

The SEI CERT C Coding Standard provides **rules** and **recommendation** from the security coding community

- **Rules** provide normative **requirements** for code
- **Recommendations** provide **guidance** to improve the **safety**, **reliability**, and **security** of software systems.

Audience: programmers

Rules are requirements: violating a rule is usually a **bug** that might be **exploited**

A violation of a recommendation does not **necessarily** indicate the presence of a defect in the code

⇒ **guidelines** for safe and secure coding

Risk assessment

An indication of

- potential **consequences** of not addressing a particular guideline
- the expected **remediation costs**

Used to **prioritize** the repair of rule violation

⇒ Violations that are more critical or less expensive will be repaired first

Each rule and recommendation has an assigned **priority**

Three values are assigned for each rule on a scale of 1 to 3 for

- **severity**
- **likelihood**
- **remediation cost**

How critical?

Severity

How **serious** are the **consequences** of the rule being ignored?

Value	Meaning	Examples of Vulnerability
1	Low	Denial-of-service attack, abnormal termination
2	Medium	Data integrity violation, information disclosure
3	High	Run arbitrary code

Likelihood

How likely is it that a flaw introduced by violating the rule can lead to an **exploitable vulnerability**?

Value	Meaning
1	Unlikely
2	Probable
3	Likely

Remediation cost

How **expensive** is it to comply with the rule?

Value	**Meaning**	**Detection**	**Correction**
1	High	Manual	Manual
2	Medium	Automatic	Manual
3	Low	Automatic	Automatic

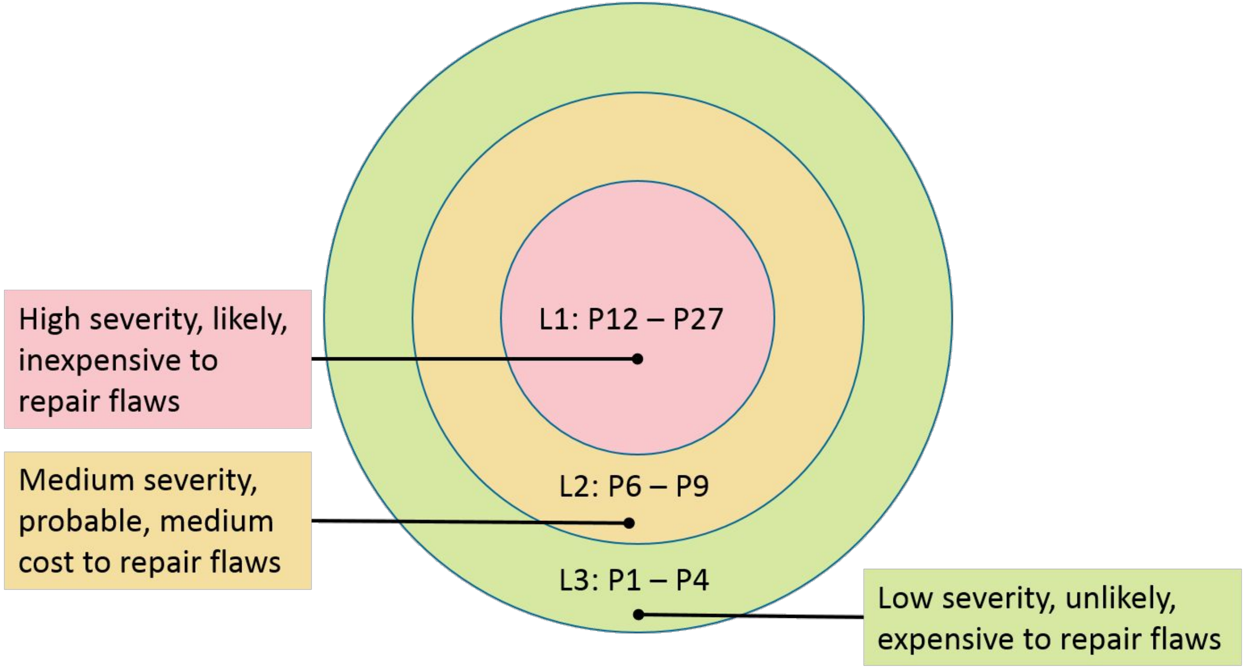
Note: Low has higher score than High ⇒ Fix low expensive issues first!

Priorities and levels

Severity, **likelihood**, and **remediation cost** are multiplied together. Product ranges from 1 to 27 with 10 possible values: 1, 2, 3, 4, 6, 8, 9, 12, 18, 27

Level	Priorities	Possible interpretation
L1	12 , 18 , 27	High severity, likely, inexpensive to repair
L2	6 , 8 , 9	Medium severity, probable, medium cost to repair
L3	1 , 2 , 3 , 4	Low severity, unlikely, expensive to repair

Priorities and levels



(picture from SEI CERT)

Rule 06. Arrays (ARR)

ARR30-C. Do not form or use out-of-bounds pointers or array subscripts

It is crucial that array indexes are always checked

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

USAGE: `if (f(i)) // use *f(i)`

```
*f(10)  == table[10]
 f(100) == NULL
```

Non compliant!

`*f(-1) == table[-1]`

⇒ No check on negative values!

Rule 06. Arrays (ARR)

Compliant version:

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

Note: Now `f(i)` is NULL if `index` is out of bound!

Rule 06. Arrays (ARR)

Alternatively, we can use a stricter type:

```
int *f(size_t index) {
  if (index < TABLESIZE) {
    return table + index;
  }
  return NULL;
}
```

Note: `size_t` is unsigned so it is enough to check that `index < TABLESIZE`

Rule 06. Arrays (ARR)

Out-of-range pointers can result in **buffer overflow**: code execution, access to sensitive information, data corruption, denial of service (**high severity**)

Overflow is **likely** to be exploitable and cannot be detected automatically in many cases (**high remediation cost**)

Severity	Likelihood	Remediation Cost	Priority	Level
High	Likely	High	**P9**	**L2**

Rule 07. Characters and Strings (STR)

STR32-C. Do not pass a non-null-terminated character sequence to a library function that expects a string

⇒ **Restricted sink**: passing a character sequence that is not null-terminated can result in accessing memory that is outside the bounds of the object

Example:

```
#include <stdio.h>
int main() {
  char c_str[3] = "abc";

  printf("%s\n", c_str);
}
```

Is the bug exploitable?

Is the previous program vulnerable?

⇒ It depends on what is after the non NULL-terminated string!

```c
int main() {
  char c_str1[3] = "abc";
  char c_str2[3] = "def";

  printf("%s\n", c_str1);
}
```

OUTPUT: `abcdef`

Fixing the code

Compliance can be achieved following recommendation STR11-C: *Do not specify the bound of a character array initialized with a string literal*

⇒ Size is computed appropriately to NULL-terminate the string!

```
#include <stdio.h>
int main() {
  char c_str[] = "abc";

  printf("%s\n", c_str);
}
```

c_str is automatically allocated as 4 bytes and string is NULL-terminated

Rule 07. Characters and Strings (STR)

Non-terminated strings can result in **buffer overflow**: code execution, access to sensitive information, data corruption, denial of service (**high severity**)

Vulnerability depends on the context and is **probable** to be exploitable and can be detected automatically in many cases (**medium remediation cost**)

Severity	Likelihood	Remediation Cost	Priority	Level
High	Probable	Medium	**P12**	**L1**

More examples

- **Rule 07. Characters and Strings (STR)**: *STR31-C. Guarantee that storage for strings has sufficient space for character data and the null terminator*
  ⇒ Typical off-by-one error!

- **Recommendation 07. Characters and Strings (STR)**: *STR07-C. Use the bounds-checking interfaces for string manipulation.* Notice that `strncpy` might leave the string unterminated
  ⇒ BSD `strlcpy` is safer!

- **Rule 10. Environment (ENV)**: *ENV33-C. Do not call system()*. Use of the `system()` function can result in exploitable vulnerabilities

String manipulation

`strcpy(dst,src)` copies `src`, including NULL, to the buffer pointed to by `dst`.

⇒ `dst` must be **large enough** to receive the copy to prevent **overflows**!

`strncpy(dst,src,n)` is similar, except that at most `n` bytes of `src` are copied

NOTE: If there is no NULL byte among the first `n` bytes of `src`, the string placed in `dst` will not be NULL-terminated!

BSD offers safer versions of these functions:

`strlcpy(dst,src,n)` copies at most n-1 bytes to `dst` and always adds a terminating NULL byte

Vulnerabilities due to `system()`

Tainted source: passing an **unsanitized** or improperly sanitized command string originating from a **tainted** source

Path resolution: If a command is specified **without a path name** and the command processor path name resolution mechanism is accessible to an attacker (path resolution might be a tainted source!)

Current working directory: If a **relative path** to an executable is specified and control over the current working directory is accessible to an attacker

Untrusted program: If the specified executable program can be **spoofed** by an attacker