An Empirical Study of C++ Vulnerabilities in Crowd-Sourced Code Examples

APPENDIX:

We will present some vulnerable examples with their types of vulnerabilities in this section. To present vulnerabilities types systematically, we included the type of vulnerability based on CWE in each example. We define each CWE in every code snippets in two section:

- **Scope of vulnerability:**
  The Scope identifies the application security area that is violated.
- **The impact of this vulnerability:**
  The Impact describes the negative technical impact that arises if an adversary succeeds in exploiting this weakness.

For listing 1:

Present CWEs:

CWE-193: Off-by-one error:

Uses an incorrect maximum or minimum value that is 1 more, or 1 less, than the correct value. The scope of this vulnerability contain Availability, integrity and confidentiality and can force the system to Dos and crash.

Scope:

- Availability
- Integrity
- Confidentiality
- System crash or DoS

Impact:

The impact of 'off-by-one' error leads to an unpredictable behavior of application. In most cases, the vulnerability results in an application crash or an infinite loop. This weakness can also lead to buffer overflow and memory corruption. In cases of a heap-based buffer overflow, the most obvious result is the application crash. If off-by-one error leads to a stack-based buffer overflow, successful code execution is more likely.

CWE-754: Improper check for unusual or exceptional conditions:

Unusual or exceptional conditions that are not expected to occur frequently during day-to-day operation of the software. Scope of these vulnerabilities on integrity, availability and force the system to DoS, crash and unexpected state.

Scope:

- Availability
- Integrity

The data which were produced as a result of a function call could be in a bad state upon return. If the return value is not checked, then this bad data may be used in operations, possibly leading to a crash or other unintended behaviors.

CWE-477: Use of obsolete function:

The code uses deprecated or obsolete functions. The scope of this vulnerability named other and the impact on quality degradation software.

CWE-1006: Bad coding practices:

Weaknesses in this category are related to coding practices that are deemed unsafe and increase the chances that an exploitable vulnerability will be present in the application. These weaknesses do not directly introduce a vulnerability, but indicate that the product has not been carefully developed or maintained. If a program is complex, difficult to maintain, not portable, or shows evidence of neglect, then there is a higher likelihood that weaknesses are buried in the code.

For listing 2:

Present CWEs:

CWE-789: Uncontrolled memory allocation:

Uses an incorrect maximum or minimum value that is 1 more, or 1 less, than the correct value. The scope of this vulnerability contain Availability, integrity and confidentiality and can force the system to Dos and crash.

Impact:

Not controlling memory allocation can result in a request for too much system memory, possibly leading to crash of the application due to out-of-memory conditions, or the consumption of a large amount of memory on the system.

CWE-252: Unchecked return value:

Unusual or exceptional conditions that are not expected to occur frequently during day-to-day operation of the software. Scope of these vulnerabilities on integrity, availability and force the system to DoS, crash and unexpected state.

Scope:

- Availability
- Integrity
- System unexpected state

An unexpected return value could place the system in a state that could lead to crash or other unintended behaviors.

CWE-476: Null pointer dereference:

Null pointer dereferences usually in the failure of the process unless exception handling (on some platforms) is available and implemented. Even when exception handling is being used, it can still be very difficult to return the software to a safe state of operation.

CWE-1006: Bad coding practices:

Described in listing 1.

For listing 3:

Sample attack scenario:

Suppose that user is asked to enter an IP address to ping and the program concatenates `ping ` with received IP address resulting in `ping 1.1.1.1`. Attacker might add `;` and a new command after `;` which will result in OS command injection.

Example input:

`ping 1.1.1.1 ; wget http://attacker.com/backdoor; ./backdoor`

Present CWEs:

CWE-78: Improper Neutralization of Special Elements used in an OS Command ('OS Command Injection'):

The software constructs all or part of an OS command using externally-influenced input from an upstream component.

Scope:

- Availability
- Integrity
- Confidentiality
- Non-reputation

Vulnerability contains confidentiality, integrity availability, non-reputation and force system Execute Unauthorized Code or Commands; DoS: Crash, Exit, or Restart; Read Files or Directories; Modify Files or Directories; Read Application Data; Modify Application Data; Hide Activities.

Impact:

Attackers could execute unauthorized commands, which could then be used to disable the software, or read and modify data for which the attacker does not have permissions to access directly. Since the targeted application is directly executing the commands instead of the attacker, any malicious activities may appear to come from the application or the application's owner.

CWE-1019: Validate Inputs:

Weaknesses in this category are related to the design and architecture of a system's input validation components. Frequently these deal with sanitizing, neutralizing and validating any externally provided inputs to minimize malformed data from entering the system and preventing code injection in the input data.

For listing 4:

Example attack scenario:

- The user sets the PATH to reference a directory under the attacker's control, such as "/my/dir/".
- The attacker creates a malicious program called "ls", and puts that program in /my/dir
- The user executes the program.
- When system() is executed, the shell consults the PATH to find the ls program.
- The program finds the attacker's malicious program, "/my/dir/ls". It doesn't find "/bin/ls" because PATH does not contain "/bin/".
- The program executes the attacker's malicious program with the raised privileges.

Present CWEs:

CWE-426: Untrusted search path:

The application searches for critical resources using an externally-supplied search path.

Scope:

- Availability
- Integrity
- Confidentiality
- Access control

Impact:

There is the potential for arbitrary code execution with privileges of the vulnerable program. The program could be redirected to the wrong files, potentially triggering a crash or hang when the targeted file is too large or does not have the expected format and the program the program could send the output of unauthorized files to the attacker.

CWE-754: Improper check for unusual or exceptional condition:

Described in Listing 1.

For listing 5:

Present CWEs:

CWE-125: Out-of-bounds Read:

The software reads data past the end, or before the beginning. Scope of this vulnerability will effect confidentiality.

Scope:

- Confidentiality

Impact:

Reading memory out of boundary can compromise program and leak data.

CWE-1019: Validate Inputs:

Described in Listing 3.

Example Attack for Listing-5:

```
int main(){
   std::vector<uint8_t> vec {0x01, 0x05};
   auto byte1 = get_from_vector<uint8_t>(vec,10);
   auto byte2 = get_from_vector<uint16_t>(vec, 20);
   auto byte4 = get_from_vector<uint32_t>(vec, 50);
   auto byte8 = get_from_vector<uint64_t>(vec, 32);
   printf("\%x - \%x - \%x - \%x",byte1,byte2,byte4,byte8);
}
```

Output:

0 - 0 - 0 - 382d3531

For Listing 7:

Present CWEs:

CWE-158: Improper Neutralization of Null Byte or NULL Character:

The software receives input from an upstream component.

Scope:

- Integrity

Impact:

As data is parsed, an injected NUL character or null byte may cause the software to believe the input is terminated earlier than it actually is, or otherwise cause the input to be misinterpreted. This could then be used to inject potentially dangerous input that occurs after the null byte or otherwise bypass validation routines.

CWE-1019: Validate Inputs:

Described in Listing 3.

In the set of 69 vulnerable answers, 42 answers were vulnerable in the latest version. There were 17 vulnerabilities in their history and 10 answers were tagged in both categories. *

By running a SourcererCC clone detection on all C++ answer posts in the Stack Overflow, and these 69 vulnerable answers, we have gotten fourteen other answers that have no link to the GitHub projects and are only vulnerable to the Stack Overflow itself.

More Description about Our Heuristic:

We extract some Keywords from a code snippet and try to make a distinct pointer that point to the code snippet.

These Keywords work together to demonstrate a code is vulnerable, Indeed we just define these keywords for vulnerable code snippets.

First, of all we check each code snippets (According to CWEs) and assess them is vulnerable or not, if it was vulnerable, we extract some keywords from code snippet and checked out as vulnerable code.

We contract to define own template and keywords usage as following:

- We have to use keywords in order; indeed, we define keywords in separate lines as following together.
- Order of keywords is important; we have to see (or not see) keywords, one after another in the code snippet. In addition, we define a threshold to check next keyword after a keyword.

Our Template have below steps:

!KEYWORD: Keywords that start with `!` should exist in source code. When multiple keywords are used, order does matter.

@!KEYWORD: Keywords that start with `@` only can come after !KEYWORD. This keyword should not exists in source code.

|!KEYWORD: If this keyword exists in source code , that's vulnerable. (Ordering not efficient on this type of keywords, and just existence is important)

Further explanation of GitHub users' beliefs about vulnerabilities in the source code is shown Figure 18:

- **The code snippet was vulnerable, and we've fixed it (or planning to fix it):**
  GitHub owner aware of the vulnerability and fixed in the source code.
- **The code snippet was vulnerable:**
  The GitHub users were aware of the vulnerability, but did not intentionally remove vulnerabilities.
- **The code snippet might've been vulnerable:**
  These users did not know about the vulnerability in the code, and they said they were reviewing the code.
- **The code snippet has been change after copying:**
  In this category, users were aware of the existence of vulnerabilities in the Stack Overflow code when copying to their project and changed it.
- **I don't know:**
  In this category, users were not aware of the existence of a vulnerability in the code, and they could not comment on the code. In this survey, one comment showed that the user was not expert in checking the code in security terms.
- **The code in question was publicly available open source code and not active on my site:**
  These Users believe that the code that used in their project cannot potentially have security vulnerability.