Analysis procedure overview

The goals of analysis are:

- To collect various numbers for test cases and/or tool reports, e.g., numbers of true tool warnings by weakness category.
- To estimate tool overlap (to what degree do tools find the same weaknesses).

For each test case, we select a subset of tool warnings. We analyze the selected warnings for correctness and find associated warnings from other tools.

Changes since SATE 2009

- Relax assumptions that a tool issues warnings based on complete knowledge of data and program flow.
- Introduce another weakness category – quality weakness

Section I describes guidelines for analysis of correctness. Section II lists guidelines for associating warnings that refer to the same (or related) weakness. Section III has guidelines for matching tool warnings to the manual findings and CVEs.

Definitions

Many weakness types, such as input validation problems, appear on a path, not on a single line. A *source* is where user input can enter a program. A *sink* is where the input is used.

I. Guidelines for analysis of correctness

Overview of correctness categories

Assign one of the following categories to each warning analyzed. Use the comment field to explain your reasons.

- True security weakness – a weakness relevant to security
- True quality weakness - poor code quality, but may not be reachable and may not be relevant to security. In other words, the issue requires developer's attention. Example: buffer overflow where input comes from the local user and the program is not run as SUID. Example: "locally true" - function has a weakness, but the function may always be called with safe parameters.
- True but insignificant weakness. Examples: database tainted during configuration, or a warning that describes properties of a standard library function without regard to its use in the code.

- Weakness status unknown - unable to determine correctness
- Not a weakness – false - an invalid conclusion about the code

The categories are ordered in the sense that a true security weakness is more important to security than a true quality weakness, which in turn is more important than a true insignificant weakness.

Next section describes the decision process for analysis of correctness, with more details for some weakness categories. This is based on our past experience and advice from experts. We consider several factors in the analysis of correctness: context, code quality, and path feasibility - we discuss these factors in the following sections.

Decision process

This section omits details about several factors (context, code quality, path feasibility) used in the decision process - see later sections for definitions and examples.

1. Mark a warning as false if any of the following holds
    - Path is clearly infeasible

    - Sink is never a problem, for example
        - Tool confuses a function call with a variable name
        - Tool misunderstands the meaning of a function, for example, tool warns that a function can return an array with less than 2 elements, when in fact the function is guaranteed to return an array with at least 2 elements.
        - Tool is confused about use of a variable, e.g., a tool warns that “an empty string is used as a password”, but the string is not used as a password at all.
        - Tool warns that object can be null, but it is always initialized.

    - For input validation issues, tool reports a weakness caused by unfiltered input, but in fact the input is filtered correctly

2. Mark a warning as insignificant if a path is not clearly infeasible, does not indicate poor code quality, and any of the following holds

    - A warning describes properties of a function (e.g., standard library function) without regard to its use in the code.
        - For example, "strncpy does not null terminate" is a true statement, but if the string is terminated after the call to strncpy in the actual use, then the warning is not significant.
    - A warning describes a property that may only lead to a security problem in unlikely (e.g., memory or disk exhaustion for a desktop or server system) and local (not caused by an external person) cases.

        - For example, a warning about unfiltered input from a command that is run only by an administrator during installation is likely insignificant.
    - A warning about coding inconsistencies (such as "unused value") does not indicate a deeper problem

3. Mark a warning as quality if

    - Poor code quality and any of the following holds:
        - Path includes infeasible conditions or values
        - Path feasibility is hard to determine
        - Code is unreachable
    - Poor code quality and not a problem under the intended security policy, but can become a problem if the policy changes (e.g., a program not intended to run with privileges is run with privileges)
        - For example, for buffer overflow, program is intended not to run with privileges (e.g., setuid) and input not under control of remote user.

4. Mark a warning as true security if path is feasible and the weakness is relevant to security

Decision details for input validation issues

1. Mark a warning as true security if input is filtered, but the filtering is not complete. This is often the case for cross-site scripting weaknesses.

Decision details for information leaks (based on Steve Christey's advice)

1. Mark these information leak warnings as false:

    - Error codes that communicate user-level errors or status. The "404 not found" error message is how the web server tells the client that the web page does not exist. (However, if the 404 message includes, for example, a full pathname, the warning is not false.)
    - Presentation information such as what color background should be used, or the font
    - process ID numbers
    - Version number of the software. (Exception: security software or other software that explicitly advertises itself as "invisible")
    - Inode numbers, file descriptor numbers, ...
    - Memory addresses (usually)
2. Mark these information leak warnings as "true security" or "true quality" or "insignificant":
    - Valid usernames
    - Passwords (encrypted or not)

- Personally identifiable information (social security number, email address, phone number, address, etc.)
- Financial information (credit card number, etc.)
- Other privacy (e.g. list of books or movies most recently purchased)
- Installation path or other internal pathnames
- Session IDs, cookies, or other mechanisms for session management
- Source code of a program that should have been executed
- Entire configuration file
- Directory listings
- Process listings
- Response discrepancy information leaks, e.g. if authentication errors can return either "username not valid" or "password not valid," this would tell the attacker whether or not a given username is valid

Whether a warning is “true security”, “true quality” or insignificant, depends on the role of the person who sees the data. For example, providing unencrypted passwords to an administrator is at least a quality problem. However, providing encrypted passwords to an administrator is probably insignificant.

To distinguish between “true security” and insignificant: if the "attacker" already has access to the targeted information or functionality through legitimate means, then it may be true-but-insignificant. Any of the program's advertised functionality counts as "legitimate." For example:

- On LinkedIn, your contacts are legitimately allowed to have access to your personal information such as email address and phone number – that would not be an information leak, and if flagged might be insignificant. But if any anonymous user can get your contact info without logging in - then it's a "true" information leak.
- A web application administrator should be expected to have access to the data files that list who the users on the system are, and probably knows the full path of the application, so that is probably insignificant.
- A denial-of-service weakness can only be exploited by an attacker with physical access to the machine. Well, that attacker can already cause DoS by throwing the machine out of a window, so this is insignificant.
- The administrator can trick the program into deleting itself. Presumably the program file already has the permissions to let the admin delete it, so this could be insignificant.

Generally, mark as insignificant or “true quality”, if the information is only available to the person who started the program, or the program is remotely available but the information is only accessible to the program's administrator.

Context

A tool does not know about context (environment and the intended security policy) for the program and may assume the worst case.

For example, if a tool reports a weakness that is caused by unfiltered input from command line or from local files, mark it as true (but it may be insignificant - see below). The reason is that the test cases are general purpose software, and we did not provide any environmental information to the participants.

Often it is necessary to determine the following:

- Who can set the environment variables
    - For web applications, the remote user
    - For desktop applications, the user who started the application
- Is the program intended to be run with privileges
- Who is the user affected by the weakness reported
    - Regular user
    - Administrator

Poor code quality vs. intended program design

A warning that points to poor code quality is usually marked as true security or true quality. On the other hand, a warning that points to code that is unusual but appropriate should be marked as insignificant.

Some examples that always indicate poor code quality:

- Not checking size of a tainted string before copying it into a buffer.
- Outputting password

Some examples that may or may not indicate poor code quality:

- Not checking for disk operation failures
- Many potential null pointer dereferences are due to the fact that methods such as malloc and strdup return null if no memory is available for allocation. If the caller does not check the result for null, this almost always leads to a null pointer dereference. However, this is not significant for some programs: if a program has run out of memory, seg-faulting is as good as anything else.
- Outputting a phone number is a serious information leak for some programs, but an intended behavior for other programs.

Memory management in Dovecot

Dovecot does memory allocation differently from other C programs. Its memory management is described here:

http://wiki.dovecot.org/Design/Memory

For example, all memory allocations (with some exceptions in data stack) return memory filled with NULLs.

This information was provided to the tool makers, so if a tool reports a warning for this intended behavior, mark it as insignificant.

Path feasibility

Determine path feasibility for a warning. Choose one of the following:

- Feasible - path shown by tool is feasible. If tool shows the sink only, the sink must be reachable.
- Feasibility difficult to determine – path is complex and contains many complicated steps involving different functions, or there are many paths from different entry points.
- Unreachable – a warning points to code within an unreachable function
- Infeasible conditions or values – a “dangerous” function is always used safely or a path is infeasible due to a flag that is set in another portion of the code.

    - An example where a function is "dangerous", but always used so that there is no problem:

```
g(int j) {
      a[j] = 'x'; // potential buffer overflow
}

… other code …

if (i < size_of_a) {
      g(i); // but g is called in a safe way
}
```

    - An example where path is infeasible due to a flag that is set elsewhere (e.g., in a different module). In the following example, tool marks NULL pointer dereference, which is infeasible because flag that is set *elsewhere in the code* is never equal FLAG_SET when value is not NULL

```
if (arg_used != NULL) { // check for NULL
      …
}
f(arg_used);

void f(int *arg_used) {
      if (value != NULL && flag == FLAG_SET) {
            *arg_used = TRUE; // NULL pointer dereference
      }
```

```
}
```

- Clearly infeasible
  - An example with infeasible path, local:

```
if (a) {
  if (!a) {
      sink
  }
}
```

  - Another example, infeasible path, local, control flow within a complete stand-alone block (e.g., a function):

```
char a[10];
if (c)
      j = 10000;
else
      j = 5;

… other code that does not change j or a …

if (!c)
      a[j] = 'x';
```

  - Infeasible path, another example

```
if (x == null && y) {
      return 0;
} else if (x == null && !y) {
      return 1;
} else {
      String parts[] = x.split(":"); // Tool reports Null pointer
      deference for x on this line – false warning because x
      cannot be null here
}
```

  - Infeasible path, for example, two functions with the same name are declared in two different classes. Tool is confused about which function is called and considers a function from the wrong class.
  - Infeasible path which shows a wrong case taken in a switch statement.

In previous SATEs, we assumed perfect understanding of code by tools, so we implicitly had only two options for path feasibility. We marked any warning for an infeasible path as false. However, poor code that is infeasible now may become feasible one day, so a warning that points to such a weakness on an infeasible path should be brought to the attention of a programmer. Additionally, analysis of feasibility for some warnings takes too much time. Therefore, we may mark some warnings on an infeasible path as quality weakness or insignificant.

II. Guidelines for associating warnings

For each tool warning in the list of selected warnings, find warnings from other tools that refer to the same (or related) weakness. For each selected warning instance, our goal is to find at least one related warning instance (if it exists) from each of the other tools. While there may be many warnings reported by a tool that are related to a particular warning, we do not need to find all of them.

Association does not produce a set of equivalent warnings; rather, it is a relationship between pairs of warnings. As such, two associated warnings may be evaluated for correctness differently.

If a warning is not in the list of selected warnings, but it was marked as associated with a selected warning, then its correctness needs to be determined.

There are several degrees of association:

- Equivalent – weakness names are the same or semantically similar; locations are the same, or in case of paths, the source and the sink are the same and the variables affected are the same.
- Strongly related – the paths are similar, where the sinks are the same conceptually (e.g., one tool may report a shorter path than another tool).
- Weakly related – warnings refer to different parts of a chain or composite; weakness names are different but related in some ways, e.g., one weakness may lead to the other, even if there is no clear chain; the paths are different but have a filter location or another important attribute in common.

More specifically, the following criteria apply to weaknesses that can be described using source-to-sink paths.

- If two warnings have the same sink, but the sources are two different variables, mark them as weakly related.
- If two warnings have the same source and sink, but paths are different, mark them as strongly related. However, if the paths involve different filters, mark them as weakly related.
- If one warning contains only the sink, and the other contains a path, the two warnings refer to the same sink and use a similar weakness name,
    - If there is no ambiguity as to which variable they refer to (and they refer to the same variable), mark them as strongly related.
    - If there are two or more variables affected and there is no way of knowing which variable the warnings refer to, mark them as weakly related.

III. Guidelines for matching warnings related to manual findings

Matching tool warnings to the manual findings is often different from matching tool warnings from different tools because the tool warnings may be at a different – lower – level than the manual findings.

Mark tool warnings related to manual findings with one or more of the following labels:

- Same instance
- Same instance, different perspective
- Same instance, different paths
    - Example: different paths, e.g., different sources, but the same sink
- Coincidental – tool reports a lower level weakness that may point the user to the high level weakness
- Other instance – tool reports a similar weakness (the same weakness type) elsewhere in the code

Due to the possibility of a large number of tool warnings related to a manual finding, we did not attempt to find all associated tool warnings for each manual finding.

Guidelines for matching warnings to CVEs are similar to the guidelines for matching warnings related to manual findings.