IPA Technical Watch
Report on Source Code Security Analysis
~ To Reduce Software Vulnerability Before Shipment ~

IT SECURITY CENTER,
INFORMATION-TECHNOLOGY PROMOTION AGENCY, JAPAN
# IPA Technical Watch: Report on Source Code Security Analysis

## Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Target Readers and Premises</td>
<td>2</td>
</tr>
<tr>
<td>Purpose of This Report</td>
<td>2</td>
</tr>
<tr>
<td>Overview</td>
<td>2</td>
</tr>
<tr>
<td>1. Introduction</td>
<td>3</td>
</tr>
<tr>
<td>1.1. Security Measures along the System Life Cycle</td>
<td>3</td>
</tr>
<tr>
<td>1.2. Current Main Security Measures</td>
<td>4</td>
</tr>
<tr>
<td>1.3. Background Factors of Non-Prevalence of Source Code Security Analysis</td>
<td>5</td>
</tr>
<tr>
<td>2. Source Code Security Analysis</td>
<td>7</td>
</tr>
<tr>
<td>2.1. Overview of Source Code Security Analysis</td>
<td>7</td>
</tr>
<tr>
<td>2.2. Types of Source Code Security Analysis Tools</td>
<td>9</td>
</tr>
<tr>
<td>2.3. Security Assessment and Characteristics</td>
<td>10</td>
</tr>
<tr>
<td>2.4. Situations Where Source Code Security Analysis Tool Is Effective</td>
<td>11</td>
</tr>
<tr>
<td>3.1. Case Study (1) ~ Case at Embedded System Vendor A</td>
<td>13</td>
</tr>
<tr>
<td>3.2. Case Study (2) ~ Case at SDNA</td>
<td>14</td>
</tr>
<tr>
<td>3.3. Case Study (3) ~ Case at IPA: Use in Tool Development</td>
<td>14</td>
</tr>
<tr>
<td>4. Trend in Source Code Security Analysis</td>
<td>15</td>
</tr>
<tr>
<td>5. Future Work of IPA</td>
<td>16</td>
</tr>
<tr>
<td>Appendix 1: Example of Source Code Security Analysis</td>
<td>18</td>
</tr>
<tr>
<td>Appendix 2: Vulnerabilities Detectable by iCodeCheckr and Their Selection Method</td>
<td>20</td>
</tr>
</tbody>
</table>
IPA Technical Watch: Report on Source Code Security Analysis
~ To Reduce Software Vulnerability Before Shipment ~

November 17, 2011
IPA (Information-technology Promotion Agency, Japan)
IT Security Center

Target Readers and Premises
This report is intended for vendors, engineers and students who are involved in software (application) development.
Source code analysis, also called source code static analysis, is categorized as a technology to detect infringements of coding rules or to detect vulnerabilities (e.g. buffer overflow). In this report, we call the technologies that are classified into the latter a “source code security analysis” technology and those that satisfy both are called a “source code analysis” technology.

Purpose of This Report
Information-technology Promotion Agency (hereinafter referred to as “IPA”) promotes vulnerability countermeasures along the system life cycle to deliver a safe and secure IT society.
This report aims to help the readers to deepen their understanding of the effectiveness and importance of source code security analysis at each implementation phase throughout the system life cycle and to spread the use of a source code security analysis tool during the actual development.

Overview
This report describes the effectiveness and importance of source code security analysis.

Chapter 1 describes the security measures along the system life cycle, the status of source code security analysis and the current situation of security measures.
Chapter 2 introduces the overview of source code security analysis and the differences from other security analysis technologies. Section 2.4 introduces the concrete situations where source code security analysis is effective.
Chapter 3 presents the case studies and success stories about source code security analysis explained up to Chapter 2 and shows its effectiveness.
Chapter 4 describes the latest trend in source code security analysis. In the near future, the use of source code security analysis tools could be required in various software development situations. IPA hopes knowing the latest trend helps the readers to introduce it to their software development.
Chapter 5 introduces IPA’s plans to promote the use of source code security analysis tools.
1. **Introduction**

1.1. **Security Measures along the System Life Cycle**

It is important to implement security countermeasures for a system as a comprehensive solution along the system life cycle from the planning to operation/use to end-of-life phase.

In this report, a system’s life cycle is divided into 6 phases: [1. Planning], [2. Design], [3. Implementation], [4. Testing], [5. Operation/Use] (and [6. End-of-Life]) (Figure 1). It is necessary to clarify the factors that could become a potential security threat and decide a security policy and countermeasures against them at each phase of the system life cycle to mitigate the risk of suffering damage.

<table>
<thead>
<tr>
<th>Security Measures</th>
</tr>
</thead>
<tbody>
<tr>
<td>Operational Measures</td>
</tr>
<tr>
<td>Vulnerability Countermeasures</td>
</tr>
<tr>
<td>Research, Trend Understanding Development Policy, Project Structure Adjustment (including business impact analysis)</td>
</tr>
<tr>
<td>Secure Programming</td>
</tr>
<tr>
<td>Source Code Security Analysis</td>
</tr>
<tr>
<td>Test (Fuzzing and other methods)</td>
</tr>
<tr>
<td>Vulnerability Assessment (Penetration Testing)</td>
</tr>
</tbody>
</table>

**Figure 1 System Life Cycle and Security Measures at Each Phase**

Security measures at each phase are addressed below.

- **[1. Planning]**
  At the [1. Planning] phase, it is important to clarify the system’s purpose, how it is used and potential security threats against the software and system to be developed, and decide a countermeasure policy and design policy for each threat based on each threat’s business impact and a countermeasure’s cost-effectiveness.

- **[2. Design]**
  At the design phase, based on the policies established at the [1. Planning] phase, the details like the functions to be implemented and data format to be defined are discussed. Keeping in mind that the system is to be used and maintained by people for real, it is important to design the system that is secure and its operation is realistic. Moreover, it is also important to think about vulnerability countermeasures before coding to prevent the software from having vulnerabilities by, for example, adopting secure programming which is know-how of how to write vulnerability-free programs.

  (Examples)
  - Designing authentication method
  - Designing logging
  - Secure programming

- **[3. Implementation]**
  At the implementation phase, security measures at the source code level are taken. As well as secure programming, source code reviews are also important.
(Examples)
• Restriction by coding rules
• Source code reviews
• Source code security analysis

• [4. Testing]
At the testing phase, vulnerability testing for the executable programs is performed while checking up their behavior at the same time. For example, there is fuzzing that is a security assessment technique to see how software behaves by creating data unexpected by the software, such as an extremely long character string, and feeding it to the software. Also, there is vulnerability assessment that inspects not only the developed software but also the overall system including the operating system and middleware.

(Examples)
• Fuzzing
• Vulnerability Assessment

• [5. Operation/Use]
Attacks that exploit vulnerabilities are evolving on a daily basis. Therefore, at the [Operation/Use] phase, it is necessary to practice vulnerability countermeasures, such as checking out information about threats and vulnerabilities, performing vulnerability assessment periodically and applying security patches as needed to fend off remote attacks.

To protect software from attacks that exploit vulnerabilities, it is essential to implement security measures at each phase.

1.2. Current Main Security Measures
When a vulnerability is found in a software or system after its shipment, sometimes system design reviews are required and there is a possibility that the cost to solve the problem could increase beyond expectation. Also, if users suffer damage because of the attacks that have exploited the vulnerability, the developer will bear the extra financial and social responsibility, not to mention the complaints from the users.

Therefore, it is especially important to have a way to detect any vulnerability unwittingly built into the software or system at an early phase of the system life cycle (before shipment) as well as a way to manage the quality.

As a security measure to prevent vulnerabilities from being built into software, adopting secure programming into the [2. Design] phase is becoming popular these days. However, the skill level of the development staff varies between individuals and therefore it is possible that some vulnerabilities are unwittingly built into the software or systems. Among the security measures to detect vulnerabilities at an early phase of the system life cycle (before shipment), performing vulnerability assessment periodically during the [4. Testing] phase and [5. Operation/Use] phase is getting established. However, vulnerability assessment is a black box testing where known attack patterns are launched to the software or system and their behavior is observed, and it cannot cover all vulnerabilities hidden in the source code.

In such situation, source code security analysis is effective. Source code security analysis is a security
measure to detect vulnerabilities hidden in the source code exhaustively, and by conducting it, it is possible to reduce vulnerabilities in the software or system before shipment. Figure 2 shows a utilization status of vulnerability assessment and source code security analysis. As shown, the developers that perform source code security analysis are about 16% and it is not as much as vulnerability assessment which is done by about 54% of the developers.

![Figure 2 Utilization Status of Source Code Security Analysis and Vulnerability Assessment](image)

1.3. Background Factors of Non-Prevalence of Source Code Security Analysis

Some presumed causes of why source code security analysis is not utilized are the following:

(1). Importance of Source Code Security Analysis Not Understood

It could be because there are few available documents by which people can confirm how important source code security analysis is. Some websites that introduce commercial products do explain its importance, but there are few documents that explain it from the third person’s standpoint. Especially, Japanese documents are so rare that management executives and project managers have little chance of understanding that source code security analysis has a great importance in fighting against vulnerabilities and is an effective vulnerability countermeasure.

In addition, the development staff is pressured with reduced development period and man-hours. Under such pressure, the development staff tends to avoid source code security analysis because it will prolong the period and increase man-hours.

(2). Difficulty of Confirming Effectiveness

Even if one tries to confirm the effectiveness of source code security analysis for real, most free tools are not provided in Japanese, making it hard to try and evaluate its effectiveness. Also, some tools may be low quality, such as being prone to false-positives or having insufficient reporting capability, for the development staff to actually use as a vulnerability countermeasure. Because of that, if the development staff has only poor knowledge about vulnerabilities, it may take a long time to judge whether a problematic line detected by the tool is indeed a vulnerability or to increase man-hours to fix the vulnerability. The fact is that there is few ways to confirm the effectiveness of source code security

---

1 Data from the questionnaire with 140 precipitants conducted at an IPA-hosted seminar in 2010.
analysis, such as cost-effectiveness, when it is actually adopted.

In the following chapters, this report explains the importance and effectiveness of source code security analysis, lack of which are the hindering factors of its non-prevalence.
2. **Source Code Security Analysis**

Source code analysis is generally called as source code static analysis as well. Source code analysis is categorized in two major types:

- **Technology to detect code that breaks the coding rules**
  A technology to see if there are coding rule violations. Its purpose is to ensure the quality of code, such as reliability, maintainability, versatility and efficiency.

- **Technology to detect vulnerability**
  A technology to see if some vulnerabilities are unwillingly built into the source code in the development process. It is a security assessment to ensure the source code’s quality of safety (source code security analysis).

This chapter focuses on the latter, source code security analysis, and describes the overview and difference from other technologies, and explains the effectiveness and importance of source code security analysis.

### 2.1. Overview of Source Code Security Analysis

Source code security analysis is a security measure to be adopted at [3. Implementation] phase to automatically detect vulnerabilities by mechanically checking out the source code that is a blueprint of software and picking out specific patterns included in the source code (Figure 3).

![Image of Source Code Security Analysis Tool](image)

**Figure 3 Image of Source Code Security Analysis Tool**

Below are some of the vulnerabilities detectable by source code security analysis tools equipped with source code security analysis technology².

- Buffer Overflow
- Format String
- Integer Overflow
- SQL Injection
- Cross-Site Scripting

---

² For concrete examples of detection, see Appendix 1.
On another front, by using a source code security analysis tool, the development staff can find problematic lines while writing software. And by judging whether found problematic lines are indeed vulnerabilities and fixing them, vulnerabilities in software can be reduced before shipment.

In this way, the development staff can reduce the risk of unexpected additional man-hours required for fixing vulnerabilities discovered after shipment (Figure 4).

As described, a source code security analysis tool helps confirm if various vulnerabilities are not hidden in source code. In addition, since a source code security analysis tool detects problematic lines by checking out specific patterns, it can find problematic lines that may not be exploited (attacked) immediately but potential vulnerabilities (that could become real vulnerabilities when additional development or some triggering event is called for) as well. On the other hand, it cannot find vulnerabilities related to functional specifications, such as vulnerabilities in access control or authentication for web applications.
Like this, it is possible to avoid building vulnerabilities into software before shipment by using a source code analysis tool. For that, it is regarded as one of the fundamental measures to counter vulnerabilities and improvement of security quality of the developed tool at the time of shipment can be expected.

2.2. Types of Source Code Security Analysis Tools

Source code security analysis tools can be divided into two major types. This report categorizes them as the source-code-entry type and the integrated development environment (IDE)-embedded type and explains each.

- **Source-Code-Entry Type**

  The source-code-entry type inspects the source code by feeding the finished source code or a set of source codes to an analysis tool (shown in Figure 3). It is effective in a case where inspecting source codes written by multiple people in a cross-codes manner. It is because it can find problematic lines that are benign in a single source code but may become vulnerabilities in combination with multiple source codes.

  In addition, most source-code-entry type tools have a characteristic that they can output a report that covers the developed software comprehensively. For that, the output is often used as a report to prove the measures are taken to reduce vulnerabilities (to ensure the software is vulnerability-free). Many of the free software available on the Internet are the source-code-entry type, although they do not have a sufficient reporting capability.

- **IDE-Embedded Type**

  The IDE-embedded type is used by embedding it into the IDE where software is actually developed. It is characteristic that vulnerabilities can be fixed immediately since it can detect problematic lines right away.

  Many current commercial source code security analysis tools are equipped with a source code security analysis function of an IDE-embedded type. For example, part of “Visual Studio 2010” and Microsoft’s IDE are equipped with a mechanism to perform source code security analysis.

  Figure 5 shows a screen image of Visual Studio 2010 after executing source code security analysis.

---

3 These types are coined for this report for the ease of understanding and are not established terms in the source code security analysis field.

As shown in Figure 5, the software may have buffer overflow vulnerability and the assessment suggests that a possible vulnerability is found in some lines.

2.3. Security Assessment and Characteristics

Besides source code security analysis, typical security assessments conducted by the developers during the system life cycle include fuzzing and vulnerability assessment.

In this chapter, the characteristics of source code security analysis, fuzzing and vulnerability assessment are described. First, the comparison of characteristics of each security assessment is summarized in Table 1.

<table>
<thead>
<tr>
<th>Analysis Technology/ [System Life Cycle Phase]</th>
<th>Necessity of Source Code</th>
<th>Types of Vulnerabilities Detectable</th>
<th>Identifying Problematic Lines</th>
<th>Confirmation of Effect</th>
<th>Identifying Type of Vulnerability</th>
</tr>
</thead>
<tbody>
<tr>
<td>Source Code Security Analysis/ [3. Implementation]</td>
<td>Necessary (White Box)</td>
<td>Wide Range of Vulnerabilities</td>
<td>Possible</td>
<td>Impossible</td>
<td>Possible</td>
</tr>
<tr>
<td>Fuzzing/ [4. Testing]</td>
<td>Unnecessary (Black Box)</td>
<td>Vulnerabilities mainly Related to Resource Management</td>
<td>Impossible</td>
<td>Possible</td>
<td>Impossible</td>
</tr>
<tr>
<td>Vulnerability Assessment/ [4 Testing]</td>
<td>Unnecessary (Black Box)</td>
<td>Limited Vulnerabilities (Mainly Those Well-Known to the Public)</td>
<td>Impossible</td>
<td>Possible</td>
<td>Possible</td>
</tr>
</tbody>
</table>

Below, based on Table 1, the characteristics of each security assessment are explained.

- **Source Code Security Analysis**

  Source code security analysis is a security assessment technique (white box testing) that takes into account the internal structure of software. The characteristics of source code security analysis are listed below.
  - Since it conduct inspection taking into account the internal structure of software, it can detect vulnerabilities hidden in rarely executed areas of the source code.
  - It is possible to easily identify where the problematic lines are in the source code.
  - It detects potential vulnerabilities that cannot be exploited immediately.
  - It cannot confirm what may possibly happen when a vulnerability is exploited.

- **Fuzzing**

  Fuzzing is a security assessment technique (black box testing) that does not take into account the internal structure of software and feeds data unexpected by the software (such as an extremely long character string) to the target source code converted into an executable program to see how the software responds. The characteristics of fuzzing are listed below.
  - Since the impact when exploited (for example, forced termination of an application) is confirmed

---

5 For example, vulnerability related to memory management is one of them.
and acknowledged, it can detect vulnerabilities that can be exploited soon.

- Because it does not take into account the internal structure of software, it is difficult to detect vulnerabilities hidden in rarely executed areas of the source code unlike source code security analysis.
- The other sections of the source code that will be affected by the impact originated from the problematic line detected need to be identified and fixed separately.

**Vulnerability Assessment**

Vulnerability assessment is a security assessment technique (black box testing) to see if the system or software has vulnerabilities by launching the known attack techniques (for example, specific patterns) to the target system or software to elicit and observe the signature response. The characteristics of vulnerability assessment are listed below:

- Unlike source code security analysis and fuzzing, it mainly targets known vulnerabilities.
- Although attack techniques (for example, specific patterns) are different, it is similar to fuzzing in some aspects, such as not taking account into the internal structure of software or observing system/software’s response.
- Since inspecting using a specific pattern and detecting a vulnerability by observing the response to the pattern, it can identify the type of the vulnerability, too.
- It cannot identify where in the source code the vulnerabilities are (if the target is a system, it is possible to identify which software contains the vulnerabilities).

Each of these vulnerability assessment techniques has advantages and disadvantages. Also, none of them can always detect all vulnerabilities. That means it is not to say that it is enough to conduct one type of security assessments. It is important to review the policy according to the nature of software and the situation, and conduct suitable security analysis.

2.4. **Situations Where Source Code Security Analysis Tool Is Effective**

In this chapter, the situations where the use of source code security analysis is effective are introduced.

- **Use for Embedded Systems and Control Systems**

  These days, embedded systems, such as smart home appliances or cell phones that connect to networks like the Internet, are growing. However, these embedded systems are said that even if the software is modified and security patches are released, it is difficult to apply security patches to these systems because they cannot be disrupted or how to apply security patch is not well known to the public. If control systems are shipped containing vulnerabilities and they are exploited, the social impact will be enormous and it could be lethal to the business for their developers.

  By using source code security analysis tools in the development of embedded systems and control systems, the developers can find vulnerabilities and fix them before shipment. As a result, it can reduce the risk that new vulnerabilities are discovered in the product and the developers struggle to fix them.

- **Use by the Developers with Poor Knowledge about Vulnerability**

  In actual development, development of some functions is often outsourced to contractors. In such case, Not all developers are knowledgeable about the vulnerability, and the software developed by an
unknown developer often has a lot of vulnerabilities.

Those vulnerabilities can be discovered through the security measures, such as vulnerability assessment or fuzzing, conducted in the [4. Testing] phase, but these assessments cannot pinpoint where the problematic lines are in the source code. There is a case that the type of the vulnerability is also unknown. Thus, even if the developers try to fix the vulnerabilities, it takes time to identify the cause in the source code. There is also a possibility that the vulnerabilities cannot be fixed.

In such situation, the use of a source code security analysis tool is effective. Since source code security analysis tools can identify where the problematic lines are, it is easier for the developers to fix them. In addition, some of commercial source code security analysis products provide the way to fix the vulnerabilities. With this, even if the developers are unknowledgeable about the vulnerability, they can fix it.

Moreover, repeating detecting and fixing the vulnerability will help the developers to build up knowledge about the vulnerability as well.

- Use for Large-Scale Software Development

Many software developers would conduct source code reviews by people to see if secure programming is practiced. As for small-scale software, it may be possible to find vulnerabilities with source code reviews by people. However, as for large-scale software, the amount of source code is huge and the cost for reviews is enormous.

Because of its size, there is a possibility that people may miss vulnerabilities that are benign in a single source code but become problematic in combination with multiple source codes.

In such situation, by using source code security analysis tools that can inspect multiple source codes together and detect problematic lines automatically, the developers can reduce the cost by a large amount.
3. Case Studies: Effective Use of Source Code Security Analysis Tools

This chapter introduces 3 case studies of the effective use of source code security analysis tools. This chapter aims to confirm the effectiveness and advantages of source code security analysis based on the concrete case studies.

3.1. Case Study (1)
~ Case at Embedded System Vender A ~

Embedded System Vender A conducts an analysis of all products to be shipped with a source code security analysis tool to find vulnerabilities. In addition, when Vender A outsources a product development to subcontractors, it provides the coding rules and mandates an assessment using a source code security analysis tool to ensure security of all products it ships out. Behind this strict conduct, Vender A has a bitter experience of receiving a report on a vulnerability in its embedded system product after shipment in the past.

When Vender A decided to implement a source code security analysis tool, it took into account the company’s actual development situation and did the following to facilitate the smooth implementation.

1). Implemented the tool at the expense of the Headquarter, not the Development Division
2). Mandated the assessment with the tool at the implementation phase
3). Held a seminar periodically to educate the development staff and raise their awareness

According to Vender A, since it mandated the assessment with the source code security analysis tool, there has been no report on critical vulnerabilities\(^6\) that may allow an attacker to execute arbitrary code on the shipped products.

Also, by using the source code security analysis tool, the vulnerabilities hidden in the source code were visualized. As a result, it produced a secondary effect in raising the security awareness of the development staff.

The development staff uses the source code security analysis tool at the implementation phase, and finds and fixes vulnerabilities before shipment. In this example, we can see that a developer can improve security quality of its products by using a source code security analysis tool. Moreover, it helps to raise the security awareness of the development staff and has produced an additional effect in them, trying not to build vulnerabilities into the products next time.

We can see the good hints for how to handle the bottlenecks in the actual situation, such as who bears the cost and how to adjust the development project structure.

---

\(^6\) The story is based on the interview with Vender A conducted by IPA.

\(^7\) Vulnerabilities related to resource management, such as buffer overflow vulnerability
3.2. Case Study (2)
~Case at SDNA~

Since 2002, Sony Digital Network Applications Inc. (SDNA) has established a dedicated security
team and has been conducting an assessment of the software it develops using a source code security
analysis tool.

SDNA reports that it was able to discover 178 vulnerabilities in 26 development projects in 2006
using the tool. The vulnerability response cost at SDNA is about a million yen per vulnerability, thus,
SDNA says it has saved about 178 million yen in total that might have been needed for
vulnerability response by using the source code analysis technology.

On another front, according to SDNA, the cost for security assessments is less than 5 percent of the
total product development cost”.

By using the source code security analysis tool at the implementation phase, SDNA finds and fixes
vulnerabilities before shipment. Moreover, by using the tool, it reduces the cost to respond to vulnerabilities
that may be found after shipment.

In this case, we can see the utilization of a source code security analysis tool will have the benefit of cost
saving, such as being able to reduce the cost to respond to vulnerability that would be found after shipment.

3.3. Case Study (3)
~Case at IPA: Use in Tool Development~

When IPA released the RFP for the “Development of the Web Attack Detection Tool iLogScanner
V3.0”, it mandated to conduct a security assessment of the developed tool as a requirement.

The contractor complied with the requirement and conducted a security assessment of the developed
tool using a commercial source code security analysis tool. As the result, the contractor found 11
problematic lines in its source code. Based on the report outputted by the source code security analysis
tool, the contractor analyzed whether those problematic lines are vulnerabilities and reported the result
to IPA.

IPA and the contractor could ensure a certain level of safety of the developed software by just
checking out the detected problematic lines.

This is a case where the contractor satisfied the security quality requirement from IPA by conducting a
source code security analysis that allowed to assess the software from the outsider’s viewpoint.

These days, the cases where the ordering parties require an assessment using a security analysis tool that
allows to inspect the software or system from the outsider’s viewpoint as an acceptance assessment
requirement are increasing (this trend will be discussed in Chapter 4). By utilizing the report outputted by a
source code security analysis tool, both sides of the contract can efficiently assure a certain level of safety of
the software.

---

4. Trend in Source Code Security Analysis

As described in Chapter 3, source code security analysis has been adopted to embedded systems and critical systems. Especially, source code security assessment will be deemed to be an inevitable requirement for the software of the systems that are facing the challenges like the following:

- Use long period of time (more than 10 years)
- Is difficult to update software (security patches)
- Require high level of security (may result in financial damage or involve human lives)

As examples, 3 movements in the industries that are facing the above issues are introduced below.

(1). Control Systems

In the field of control systems, security awareness is on the rise partially due to the Stuxnet virus that caused quite a stir in 2010. There are active movements to develop a security standard for control systems and establish an evaluation and certification scheme based on the standard. Examples of such movements are IEC62443 and ISASecure EDSA (Embedded Device Security Assurance). They define the issues such as security functions, development processes and test specifications of the embedded systems. ISASecure EDSA is one step ahead and it lists a source code analysis (static analysis) during the development process and compliance with the secure coding rules as requirements for certification depending on the certification level.

(2). Functional Safety Standard (IEC61508)

Functional safety is a standard to ensure and improve safety of the systems and will affect a wide range of industries. In the automotive industry, ISO26262 is the one. For accreditation, some commercial source code analysis tools are accredited to assess the compliance with the rules.

(3). Credit Card Industry (PCI DSS : Payment card industry data security standard)

In PCI DSS Ver2.0, the Requirement 6.3.2 demands to “review of custom code prior to release to production or customers in order to identify any potential coding vulnerability”.

NOTE: PCI DSS adds that “this requirement for code reviews applies to all custom code (both internal and public-facing), as part of the system development life cycle.” To comply with the requirement, it could be possible to conduct the review by people, but with large-scale software, the use of an automated analysis tool is a more effective choice.

As seen above, source code analysis is getting adopted into the standards or industry guidelines.

Cost increase for the development is a immediate challenge but it is important to consider the utilization of source code analysis from the perspective of the total cost the company will pay throughout the life cycle of software development, operation and maintenance.
5. Future Work of IPA

IPA has been promoting vulnerability countermeasures along the system life cycle to deliver a safe and secure IT society and providing the various contents (such as guides, reports and tools) tailored to each phase of the system life cycle.

IPA believes that by promoting the wide use of source code security analysis explained in this report, IPA can contribute to the realization of a safe and secure IT society.

With that, IPA has released a tool called “iCodeChecker” that can test the importance and effectiveness of source code security analysis. iCodeChecker focuses on the education about the effectiveness and importance of the source code security analysis technology. IPA expects it to be used by the software development staff and educational institutions.

IPA hopes that it will help people understand the usefulness and effectiveness of source code security analysis and be adopted into the actual software development process by the development staff.

- Characteristics of iCodeChecker
  A concept image of iCodeChecker is shown in Figure 7.

---

The characteristics\(^{11}\) of iCodeChecker are the following:

1. **Detect the vulnerabilities often unwittingly built into software and are highly dangerous**
   
   It limits the target vulnerabilities it can detect to those that could allow an attacker to execute arbitrary code\(^{12}\). Compared to general commercial products or tools, the vulnerabilities it can detect are limited, but those that would cause a significant impact when exploited, such as arbitrary code execution, are expected to be detected.

2. **Target the programming language highly used**
   
   It targets the C programming language (ANSI C), which is highly used\(^{13}\) and easy to write vulnerabilities with. The C is often used in the development of the embedded systems that are said to be difficult to apply security patches to fix vulnerabilities. By showing the effectiveness of the source code security analysis technology to the development staff, it is expected to lead to the establishment of a safe and secure IT society.

3. **Have a detailed reporting function**
   
   It has a detailed reporting function that open source software tools do not provide. By using this function, it can not only detect vulnerabilities but also provide the information about where the vulnerabilities are in the source code and how to fix them. The users can fix the problematic lines based on the report contents.

4. **Provide Easy to Use Analysis Function**
   
   The way to use iCodeChecker and procedures are simplified. By distributing iCodeChecker as a virtual machine image to make it technically easier to install it or preparing a simple web interface, it is designed to be easily usable for non-technical users at educational institutions.

---

\(^{11}\) These are as of the writing of this report and are subject to change.

\(^{12}\) For the vulnerabilities detectable by the Tool, see Appendix 2.

Appendix 1: Example of Source Code Security Analysis

This chapter explains how a source code security analysis tool works using some concrete examples. For example, in the case of the C programming language, the fundamental basis is like the following:

```c
#define HEADER_SIZE 8

int vuln_func(char *data, int dlen)
{
    char *buf;
    int size;

    if (dlen == 0)
        return -1;

    size = dlen + HEADER_SIZE;
    if (!(buf = malloc(size)))
        return -1;

    memset(buf, 0, HEADER_SIZE);
    memcpy(buf + HEADER_SIZE, data, dlen);

    return handle_data(buf, size);
}
```

Appendix Figure 1 Example of Vulnerable Source Code

1). The `vuln_func` function as a subject of analysis, the tool starts the assessment of its internal process (Line: 3).
2). The tool acknowledges a function’s parameter `dlen` is a signed integer variable (Line: 4, 5).
3). The tool acknowledges the possibility of integer overflow seeing that `HEADER_SIZE` is added to `dlen` (Line: 11). In fact, if a value to be set to `dlen` is `0xffffffff`, integer overflow occurs and the value results in `0` (if `0xffffffff+1`, it results in `1`).
4). The tool acknowledges the program calls the `malloc` function to allocate memory using a variable `dlen` that is vulnerable to integer overflow as its parameter (Line: 12), and acknowledges there is a possibility that the buffer size to be allocated by the `malloc` function is improper.
5). The tool acknowledges the program calls the `memset` function and `memcpy` function using a buffer `buf` whose allocated size may be improper (Line: 15, 16), and acknowledges the possibility of buffer overflow because the buffer size would be wrong.
6). The tool acknowledges the program calls the `handle_data` function using the problematic variable `buf`

---

14 It is a concept image of how the tool works and the actual behavior may differ depending on the specifications of a source code security analysis tool to be used.
and size as its parameters, and moves on to inspect the handle_data function (Line: 18).

Like this, based on the input for and internal process of an assessment subject (the vuln_func function), the tool inspects and see if there is a possibility that any problem may occur.

In an example above, a problem originates at the step 3) and is carried on into the step 4) to 6). From the security viewpoint, there is a possibility that buffer overflow may occur at the step 5). However, which step from 3) to 5) a source code security analysis tool would actually detect as a problem line differs depending on the functions and specifications of the tool.
Appendix 2: Vulnerabilities Detectable by iCodeCheckr\textsuperscript{15} and Their Selection Method

This chapter explains the vulnerabilities detectable by iCodeChecker and the selection method behind.

IPA thought that iCodeChecker does not need to detect all vulnerabilities exhaustively from its standpoint of education of source code security analysis,

With that, IPA decided that it would select the target vulnerabilities that iCodeChecker can detect based on the classifications of vulnerability types defined by CWE (Common Weakness Enumeration)\textsuperscript{16} maintained by MITRE in the U.S. In CWE, the vulnerability types are classified into 4 groups: View, Category, Weakness and Compound Element. The Version 2.1, the latest version as of writing of this report, lists the total of 777 vulnerabilities with 27 as View elements, 157 as Category elements, 693 as Weakness elements and 9 as Compound Element elements.

From 693 Weakness vulnerability types listed in CWE, IPA selected those iCodeChecker can detect based on the following requirements:

1. The critical vulnerability types that may lead to arbitrary code execution
   Using the CWE list as reference, IPA narrowed down to and picked up those that may lead to arbitrary code execution which could cause a significant impact when exploited.

2. The vulnerability types the developers often unwittingly write into software
   Using the Vulnerability Type Distributions in CVE\textsuperscript{17} report released by MITRE and the Vulnerability Countermeasure Information Database JVN iPedia Registration Status reports\textsuperscript{18} as references, IPA selected those that the developers tend to unwittingly write into software.

In addition to the above, IPA reviewed the vulnerability types from the standpoint of desirability to be detected by iCodeChecker and narrowed down to the following 6 types (Appendix Table 1).

\begin{table}[h]
\centering
\begin{tabular}{|c|c|}
\hline
No. & Vulnerability Type & CWE Identifier \\
\hline
1 & Buffer Copy without Checking Size of Input & CWE-120 \\
2 & Improper Validation of Array Index & CWE-129 \\
3 & Use of sizeof\((\text{\texttt{f}})\) on a Pointer Type & CWE-467 \\
4 & Uncontrolled Format String & CWE-134 \\
5 & Integer Overflow or Wraparound & CWE-190 \\
6 & Integer Underflow (Wrap or Wraparound)\textsuperscript{19} & CWE-191 \\
7 & Signed to Unsigned Conversion Error & CWE-195 \\
8 & Use of Uninitialized Variable\textsuperscript{20} & CWE-457 \\
\hline
\end{tabular}
\caption{Vulnerabilities Detectable by iCodeChecker}
\end{table}

Although the number of vulnerabilities iCodeChecker can detect is limited to 8 types, IPA believes that since many critical vulnerabilities are expected to be detected, it is effective as an educational tool to show the effectiveness and importance of a source code security analysis tool.

\textsuperscript{15} They are as the time of this writing and subject to change.
\textsuperscript{16} CWE Overview, IPA, \url{http://www.ipa.go.jp/security/vuln/CWE.html}
\textsuperscript{17} Vulnerability Type Distributions in CVE, MITRE, \url{http://www.cve.mitre.org/docs/vuln-trends/index.html}
\textsuperscript{18} Vulnerability Countermeasure Information Database JVN iPedia Registration Status, IPA, \url{http://www.ipa.go.jp/security/vuln/report/press.html#JVNiPedia}
\textsuperscript{19} Added after the publication of the Japanese version of this report.
\textsuperscript{20} Added after the publication of the Japanese version of this report.