FIO30-C. Exclude user input from format strings

Never call a formatted I/O function with a format string containing a tainted value. An attacker who can fully or partially control the contents of a format string can crash a vulnerable process, view the contents of the stack, view memory content, or write to an arbitrary memory location. Consequently, the attacker can execute arbitrary code with the permissions of the vulnerable process [Seacord 2013b]. Formatted output functions are particularly dangerous because many programmers are unaware of their capabilities. For example, formatted output functions can be used to write an integer value to a specified address using the %p conversion specifier.

Noncompliant Code Example

The incorrect_password() function in this noncompliant code example is called during identification and authentication to display an error message if the specified user is not found or the password is incorrect. The function accepts the name of the user as a string referenced by user. This is an exemplar of untrusted data that originates from an unauthenticated user. The function constructs an error message that is then output to stderr using the C Standard fprintf() function.

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

void incorrect_password(const char *user) {
    /* User names are restricted to 256 or fewer characters */
    static const char msg_format[] = "%s cannot be authenticated.\n";
    size_t len = strlen(user) + sizeof(msg_format);
    char *msg = (char *)malloc(len);
    if (msg == NULL) {
        /* Handle error */
    }
    ret = snprintf(msg, len, msg_format, user);
    if (ret < 0) {
        /* Handle error */
    } else if (ret >= len) {
        /* Handle truncated output */
    }
    fprintf(stderr, msg);
    free(msg);
}
```

The incorrect_password() function calculates the size of the message, allocates dynamic storage, and then constructs the message in the allocated memory using the snprintf() function. The addition operations are not checked for integer overflow because the string referenced by user is known to have a length of 256 or less. Because the %s characters are replaced by the string referenced by user in the call to snprintf(), the resulting string needs 1 byte less than is allocated. The snprintf() function is commonly used for messages that are displayed in multiple locations or messages that are difficult to build. However, the resulting code contains a format-string vulnerability because the msg includes untrusted user input and is passed as the format-string argument in the call to fprintf().

Compliant Solution (fputs())

This compliant solution fixes the problem by replacing the fprintf() call with a call to fputs(), which outputs msg directly to stderr without evaluating its contents:
```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

void incorrect_password(const char *user) {
    int ret;
    /* User names are restricted to 256 or fewer characters */
    static const char msg_format[] = "%s cannot be authenticated.\n";
    size_t len = strlen(user) + sizeof(msg_format);
    char *msg = (char *)malloc(len);
    if (msg == NULL) {
        /* Handle error */
    } else {
        ret = snprintf(msg, len, msg_format, user);
        if (ret < 0) {
            /* Handle error */
        } else if (ret >= len) {
            /* Handle truncated output */
        }
        fputs(msg, stderr);
        free(msg);
    }
}
```

**Compliant Solution (\printf())**

This compliant solution passes the untrusted user input as one of the variadic arguments to \printf() and not as part of the format string, eliminating the possibility of a format-string vulnerability:

```c
#include <stdio.h>

void incorrect_password(const char *user) {
    static const char msg_format[] = "%s cannot be authenticated.\n";
    fprintf(stderr, msg_format, user);
}
```

**Noncompliant Code Example (POSIX)**

This noncompliant code example is similar to the first noncompliant code example but uses the POSIX function syslog() [IEEE Std 1003.1-2013] instead of the \printf() function. The syslog() function is also susceptible to format-string vulnerabilities.

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <syslog.h>

void incorrect_password(const char *user) {
    int ret;
    /* User names are restricted to 256 or fewer characters */
    static const char msg_format[] = "%s cannot be authenticated.\n";
    size_t len = strlen(user) + sizeof(msg_format);
    char *msg = (char *)malloc(len);
    if (msg == NULL) {
        /* Handle error */
    } else {
        ret = snprintf(msg, len, msg_format, user);
        if (ret < 0) {
            /* Handle error */
        } else if (ret >= len) {
            /* Handle truncated output */
        }
        syslog(LOG_INFO, msg);
        free(msg);
    }
}
```
The `syslog()` function first appeared in BSD 4.2 and is supported by Linux and other modern UNIX implementations. It is not available on Windows systems.

**Compliant Solution (POSIX)**

This compliant solution passes the untrusted user input as one of the variadic arguments to `syslog()` instead of including it in the format string:

```c
#include <syslog.h>

void incorrect_password(const char *user) {
    static const char msg_format[] = "%s cannot be authenticated.\n";
    syslog(LOG_INFO, msg_format, user);
}
```

**Risk Assessment**

Failing to exclude user input from format specifiers may allow an attacker to crash a vulnerable process, view the contents of the stack, view memory content, or write to an arbitrary memory location and consequently execute arbitrary code with the permissions of the vulnerable process.

<table>
<thead>
<tr>
<th>Rule</th>
<th>Severity</th>
<th>Likelihood</th>
<th>Remediation Cost</th>
<th>Priority</th>
<th>Level</th>
</tr>
</thead>
<tbody>
<tr>
<td>FIO30-C</td>
<td>High</td>
<td>Likely</td>
<td>Medium</td>
<td>P18</td>
<td>L1</td>
</tr>
</tbody>
</table>

**Automated Detection**

<table>
<thead>
<tr>
<th>Tool</th>
<th>Version</th>
<th>Checker</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Astrée</td>
<td>20.10</td>
<td></td>
<td>Supported via stubbing/taint analysis</td>
</tr>
<tr>
<td>Axivion Bauhaus Suite</td>
<td>7.2.0</td>
<td>CertC-FIO30</td>
<td>Partially implemented</td>
</tr>
<tr>
<td>CodeSonar</td>
<td>6.0p0</td>
<td>IO.INJ.FMT, MISC.FMT</td>
<td>Format string injection Format string</td>
</tr>
<tr>
<td>Compass/ROSE</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Coverity</td>
<td>2017.07</td>
<td>TAIANTED_STRING</td>
<td>Implemented</td>
</tr>
<tr>
<td>GCC</td>
<td>4.3.5</td>
<td></td>
<td>Can detect violations of this rule when the <code>-Wformat-security</code> flag is used</td>
</tr>
<tr>
<td>Helix QAC</td>
<td>2021.2</td>
<td>C4916, C4917, C4918, C++4916, C++4917, C++4918</td>
<td></td>
</tr>
<tr>
<td>Klocwork</td>
<td>2021.1</td>
<td>SV.FMTSTR.GENERIC, SV.TAINTED.FMTSTR</td>
<td></td>
</tr>
<tr>
<td>LDRA tool suite</td>
<td>9.7.1</td>
<td>86 D</td>
<td>Partially Implemented</td>
</tr>
<tr>
<td>Parasoft C/C++test</td>
<td>2021.1</td>
<td>CERT_C-FIO30-a, CERT_C-FIO30-b, CERT_C-FIO30-c</td>
<td>Avoid calling functions printf/fprintf with only one argument other than string constant. Avoid using functions fprintf/fprintf with only two parameters, when second parameter is a variable. Never use unfiltered data from an untrusted user as the format parameter</td>
</tr>
<tr>
<td>PC-lint Plus</td>
<td>1.4</td>
<td>592</td>
<td>Partially supported: reports non-literal format strings</td>
</tr>
<tr>
<td>Polyspace Bug Finder</td>
<td>R2021a</td>
<td>CERT C: Rule FIO30-C</td>
<td>Checks for tainted string format (rule partially covered)</td>
</tr>
<tr>
<td>PRQA QA-C</td>
<td>9.7</td>
<td>4916, 4917, 4918</td>
<td></td>
</tr>
<tr>
<td>PRQA QA-C++</td>
<td>4.4</td>
<td>4916, 4917, 4918</td>
<td></td>
</tr>
<tr>
<td>PVS-Studio</td>
<td>7.13</td>
<td>V618</td>
<td></td>
</tr>
<tr>
<td>Splint</td>
<td>3.1.1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Related Vulnerabilities**

Two examples of format-string vulnerabilities resulting from a violation of this rule include Ettercap and Samba.
In Ettercap v.NG-0.7.2, the ncurses user interface suffers from a format-string defect. The curses_msg() function in ec_curses.c calls wdg_scroll_print(), which takes a format string and its parameters and passes it to vw_printw(). The curses_msg() function uses one of its parameters as the format string. This input can include user data, allowing for a format-string vulnerability.

The Samba AFS ACL mapping VFS plug-in fails to properly sanitize user-controlled file names that are used in a format specifier supplied to snprintf(). This security flaw becomes exploitable when a user can write to a share that uses Samba's afsacl.so library for setting Windows NT access control lists on files residing on an AFS file system.

Search for vulnerabilities resulting from the violation of this rule on the CERT website.

Related Guidelines

<table>
<thead>
<tr>
<th>Taxonomy</th>
<th>Taxonomy item</th>
<th>Relationship</th>
</tr>
</thead>
<tbody>
<tr>
<td>CERT Oracle Secure Coding Standard for Java</td>
<td>IDS06-J. Exclude unsanitized user input from format strings</td>
<td>Prior to 2018-01-12: CERT: Unspecified Relationship</td>
</tr>
<tr>
<td>CERT Perl Secure Coding Standard</td>
<td>IDS30-PL. Exclude user input from format strings</td>
<td>Prior to 2018-01-12: CERT: Unspecified Relationship</td>
</tr>
<tr>
<td>CWE 2.11</td>
<td>CWE-134, Uncontrolled Format String</td>
<td>2017-05-16: CERT: Exact</td>
</tr>
<tr>
<td>CWE 2.11</td>
<td>CWE-20, Improper Input Validation</td>
<td>2017-05-17: CERT: Rule subset of CWE</td>
</tr>
</tbody>
</table>

Bibliography

[IEEE Std 1003.1:2013] XSH, System Interfaces, syslog
[Seacord 2013b] Chapter 6, "Formatted Output"
[Viega 2005] Section 5.2.23, "Format String Problem"