POS33-C. Do not use vfork()

Using the vfork function introduces many portability and security issues. There are many cases in which undefined and implementation-specific behavior can occur, leading to a denial-of-service vulnerability.

According to the vfork man page,

> The vfork() function has the same effect as fork(), except that the behavior is undefined if the process created by vfork() either modifies any data other than a variable of type pid_t used to store the return value from vfork(), or returns from the function in which vfork() was called, or calls any other function before successfully calling _exit() or one of the exec family of functions.

Furthermore, older versions of Linux are vulnerable to a race condition, occurring when a privileged process calls vfork(), and then the child process lowers its privileges and calls execve(). The child process is executed with the unprivileged user's UID before it calls execve().

Because of the implementation of the vfork() function, the parent process is suspended while the child process executes. If a user sends a signal to the child process, delaying its execution, the parent process (which is privileged) is also blocked. This means that an unprivileged process can cause a privileged process to halt, which is a privilege inversion resulting in a denial of service.

This code example shows how difficult it is to use vfork() without triggering undefined behavior. The lowering of privileges in this case requires a call to setuid(), the behavior of which is undefined because it occurs between the vfork() and the execve().

```c
pid_t pid = vfork();
if (pid == 0) /* child */ {
    setuid(unprivileged_user); /* undefined behavior */
    /* Window of vulnerability to privilege inversion on older versions of Linux */
    if (execve(filename, NULL, NULL) == -1) {
        /* Handle error */
    }
    /* In normal operations, execve() might fail; if it does, vfork() behavior is undefined. */
    _exit(1); /* in case execve() fails */
}
```

Use fork() instead of vfork() in all circumstances.

Noncompliant Code Example

This noncompliant code example calls vfork() and then execve(). As previously discussed, a vfork()/execve() pair contains an inherent race window on some implementations.

```c
char *filename = /* something */;

pid_t pid = vfork();
if (pid == 0) /* child */ {
    if (execve(filename, NULL, NULL) == -1) {
        /* Handle error */
    }
    _exit(1); /* in case execve() fails */
}
```

Compliant Solution

This compliant solution replaces the call to vfork() with a call to fork(), which does not contain a race condition, and eliminates the denial-of-service vulnerability:
char *filename = /* something */;

pid_t pid = fork();
if (pid == 0) /* child */ {
    if (execve(filename, NULL, NULL) == -1) {
        /* Handle error */
    }
    _exit(1); /* in case execve() fails */
}

Risk Assessment

Using the `vfork` function can result in a denial-of-service vulnerability.

<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>POS33-C</td>
<td>low</td>
<td>probable</td>
<td>low</td>
<td>P6</td>
<td>L2</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>Axivion Bauhaus Suite</td>
<td>6.9.0</td>
<td>CertC-POS33</td>
<td></td>
</tr>
<tr>
<td>CodeSonar</td>
<td>5.2p0</td>
<td>BADFUNC.VFORK</td>
<td>Use of <code>vfork</code></td>
</tr>
<tr>
<td>Coverity</td>
<td>2017.07</td>
<td>DONTCALL</td>
<td>Implemented</td>
</tr>
<tr>
<td>LDRA tool suite</td>
<td>9.7.1</td>
<td>44 S</td>
<td>Fully implemented</td>
</tr>
<tr>
<td>Compass/ROSE</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Parasoft C/C++-test</td>
<td>10.4.2</td>
<td>CERT_C-POS33-a</td>
<td>Avoid using the 'vfork()' function</td>
</tr>
<tr>
<td>Polyspace Bug Finder</td>
<td>R2020a</td>
<td>CERT C: Rule POS33-C</td>
<td>Checks for use of obsolete standard function (rule fully covered)</td>
</tr>
<tr>
<td>PRQA QA-C</td>
<td>9.7</td>
<td>5023</td>
<td>Fully implemented</td>
</tr>
<tr>
<td>SonarQube C/C++ Plugin</td>
<td>3.11</td>
<td>ObsoletePosixFunction</td>
<td></td>
</tr>
</tbody>
</table>

Related Vulnerabilities

Search for vulnerabilities resulting from the violation of this rule on the [CERT website](https://csrc.nist.gov/safesearch).  

Related Guidelines

* Key here (explains table format and definitions)

<table>
<thead>
<tr>
<th>Taxonomy</th>
<th>Taxonomy item</th>
<th>Relationship</th>
</tr>
</thead>
<tbody>
<tr>
<td>CWE 2.11</td>
<td>CWE-242, Use of inherently dangerous function</td>
<td>2017-07-05: CERT: Rule subset of CWE</td>
</tr>
<tr>
<td>MITRE CWE</td>
<td>CWE-676</td>
<td>Prior to 2018-01-12:CERT:Relationship Unspecified</td>
</tr>
<tr>
<td>CWE 3.1</td>
<td>CWE-676, Use of Potentially Dangerous Function</td>
<td>2018-10-18:CERT:None</td>
</tr>
</tbody>
</table>

CERT-CWE Mapping Notes

* Key here for mapping notes

**CWE-242 and POS33-C**

CWE-242 = Union( POS33-C, list) where list =

* Use of dangerous functions besides `vfork`

**CWE-676 and POS33-C**
INTERSECTION(CWE-676, POS33-C) =

Bibliography

[Wheeler 2003] Section 8.6