INT04-C. Enforce limits on integer values originating from tainted sources

All integer values originating from tainted sources should be evaluated to determine if they have identifiable upper and lower bounds. If so, these limits should be enforced by the interface. Restricting the input of excessively large or small integers helps prevent overflow, truncation, and other type range errors. Furthermore, it is easier to find and correct input problems than it is to trace internal errors back to faulty inputs.

Noncompliant Code Example

In this noncompliant code example, length is the value of a user-defined (and thus potentially untrusted) environment variable whose value is used to determine the size of a dynamically allocated array, table. In compliance with INT30-C. Ensure that unsigned integer operations do not wrap, the code prevents unsigned integer wrapping but does not impose any upper bound on the size of the array, making it possible for the user to cause the program to use an excessive amount of memory.

```c
char** create_table(void) {
    const char* const lenstr = getenv("TABLE_SIZE");
    const size_t length = lenstr ? strtoul(lenstr, NULL, 10) : 0;
    if (length > SIZE_MAX / sizeof(char *))
        return NULL; /* Indicate error to caller */
    const size_t table_size = length * sizeof(char *);
    char** const table = (char **)malloc(table_size);
    if (table == NULL)
        return NULL; /* Indicate error to caller */
    /* Initialize table... */
    return table;
}
```

Because length is user controlled, the value can result in a large block of memory being allocated or can cause the call to malloc() to fail. Depending on how error handling is implemented, it may result in a denial-of-service attack or other error.

Compliant Solution

This compliant solution defines the acceptable range for length as [1, MAX_TABLE_LENGTH]. The length parameter is declared as size_t, which is unsigned by definition. Consequently, it is not necessary to check length for negative values (see INT01-C. Use rsize_t or size_t for all integer values representing the size of an object).

```c
enum { MAX_TABLE_LENGTH = 256 }; 
char** create_table(void) {
    const char* const lenstr = getenv("TABLE_SIZE");
    const size_t length = lenstr ? strtoul(lenstr, NULL, 10) : 0;
    if (length == 0 || length > MAX_TABLE_LENGTH)
        return NULL; /* Indicate error to caller */
    const size_t table_size = length * sizeof(char *);
    char** const table = (char **)malloc(table_size);
    if (table == NULL)
        return NULL; /* Indicate error to caller */
    /* Initialize table... */
    return table;
}
```

The test for length = 0 ensures that a nonzero number of bytes is allocated. (See MEM04-C. Beware of zero-length allocations.)

Noncompliant Code Example

In this noncompliant example, the tainted integer color_index is used in pointer arithmetic to index into the array table:
const char *table[] = { "black", "white", "blue", "green" };

const char *set_background_color(void) {
    int color_index;
    GET_TAINTED_INTEGER(int, color_index);

    const char *color = table[color_index];  /* Violation */
    /* ... */
    return color;
}

Compliant Solution

This compliant solution defines the acceptable range for `color_index` as [1, MAX_COLOR_INDEX]:

```
enum { MAX_COLOR_INDEX = 3 };
const char *table[] = { "black", "white", "blue", "green" };

const char *set_background_color(void) {
    int color_index;
    GET_TAINTED_INTEGER(int, color_index);

    if (color_index < 0 || color_index > MAX_COLOR_INDEX) {
        return NULL;  /* Indicate error to caller */
    }

    const char *color = table[color_index];
    /* ... */
    return color;
}
```

Noncompliant Code Example (Heartbleed)

CERT vulnerability 720951 describes a vulnerability in OpenSSL versions 1.0.1 through 1.0.1f, popularly known as "Heartbleed." This vulnerability allows an attacker to steal information that under normal conditions would be protected by Secure Socket Layer/Transport Layer Security (SSL/TLS) encryption.

Despite the seriousness of the vulnerability, Heartbleed is the result of a common programming error and an apparent lack of awareness of secure coding principles. Following is the vulnerable code:

```
const char *table[] = { "black", "white", "blue", "green" };

const char *set_background_color(void) {
    int color_index;
    GET_TAINTED_INTEGER(int, color_index);

    const char *color = table[color_index];  /* Violation */
    /* ... */
    return color;
}``
int dtls1_process_heartbeat(SSL *s) {
    unsigned char *p = &s->s3->rrec.data[0], *pl;
    unsigned short hbtype;
    unsigned int payload;
    unsigned int padding = 16; /* Use minimum padding */

    /* Read type and payload length first */
    hbtype = *p++;
    n2s(p, payload);
    pl = p;

    /* ... More code ... */

    if (hbtype == TLS1_HB_REQUEST) {
        unsigned char *buffer, *bp;
        int r;

        /* Allocate memory for the response, size is 1 byte
         * message type, plus 2 bytes payload length, plus
         * payload, plus padding
         */
        buffer = OPENSSL_malloc(1 + 2 + payload + padding);
        bp = buffer;

        /* Enter response type, length and copy payload */
        *bp++ = TLS1_HB_RESPONSE;
        s2n(payload, bp);
        memcpy(bp, pl, payload);

        /* ... More code ... */
    }
    /* ... More code ... */
}

This code processes a "heartbeat" packet from a client. As specified in RFC 6520, when the program receives a heartbeat packet, it must echo the packet's data back to the client. In addition to the data, the packet contains a length field that conventionally indicates the number of bytes in the packet data, but there is nothing to prevent a malicious packet from lying about its data length.

The `p` pointer, along with `payload` and `pl`, contain data from a packet. The code allocates a `buffer` sufficient to contain `payload` bytes, with some overhead, then copies `payload` bytes starting at `pl` into this buffer and sends it to the client. Notably absent from this code are any checks that the `payload` integer variable extracted from the heartbeat packet corresponds to the size of the packet data. Because the client can specify an arbitrary value of `payload`, an attacker can cause the server to read and return the contents of memory beyond the end of the packet data, which violates INT04-C: Enforce limits on integer values originating from tainted sources. The resulting call to `memcpy()` can then copy the contents of memory past the end of the packet data and the packet itself, potentially exposing sensitive data to the attacker. This call to `memcpy()` violates ARR38-C: Guarantee that library functions do not form invalid pointers. A version of ARR38-C also appears in ISO/IEC TS 17961:2013, "Forming invalid pointers by library functions [libptr]." This rule would require a conforming analyzer to diagnose the Heartbleed vulnerability.

Compliant Solution (Heartbleed)

OpenSSL version 1.0.1g contains the following patch, which guarantees that `payload` is within a valid range. The range is limited by the size of the input record.
int dtls1_process_heartbeat(SSL *s) {
    unsigned char *p = &s->s3->rrec.data[0], *pl;
    unsigned short hbtype;
    unsigned int payload;
    unsigned int padding = 16; /* Use minimum padding */

    /* ... More code ... */

    /* Read type and payload length first */
    if (1 + 2 + 16 > s->s3->rrec.length)
        return 0; /* silently discard */
    hbtype = *p++;
    n2s(p, payload);
    if (1 + 2 + payload + 16 > s->s3->rrec.length)
        return 0; /* silently discard per RFC 6520 */
    pl = p;

    /* ... More code ... */

    if (hbtype == TLS1_HB_REQUEST) {
        unsigned char *buffer, *bp;
        int r;

        /* Allocate memory for the response, size is 1 byte
         * message type, plus 2 bytes payload length, plus
         * payload, plus padding */
        buffer = OPENSSL_malloc(1 + 2 + payload + padding);
        bp = buffer;
        /* Enter response type, length and copy payload */
        *bp++ = TLS1_HB_RESPONSE;
        s2n(payload, bp);
        memcpy(bp, pl, payload);
        /* ... More code ... */
    }

    /* ... More code ... */
}

Risk Assessment

Failing to enforce the limits on integer values can result in a denial-of-service attack, unauthorized disclosure of information, or to run arbitrary code.

<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>INT04-C</td>
<td>High</td>
<td>Probable</td>
<td>High</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>CodeSonar</td>
<td>5.2p0</td>
<td>IO.TAINT.SIZE</td>
<td>Tainted allocation size</td>
</tr>
<tr>
<td></td>
<td></td>
<td>LANG.MEM.TBA</td>
<td>Tainted buffer access</td>
</tr>
<tr>
<td></td>
<td></td>
<td>IO.TAINT.ADDR</td>
<td>Tainted network address</td>
</tr>
<tr>
<td></td>
<td></td>
<td>IO.UT.HOST</td>
<td>Untrusted Network Host</td>
</tr>
<tr>
<td></td>
<td></td>
<td>IO.UT.PORT</td>
<td>Untrusted Network Port</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(general)</td>
<td>CodeSonar will track the tainted value, along with any limits applied to it, and flag any problems caused by underconstraint. Warnings of a wide range of classes may be triggered, including tainted allocation size, buffer overrun, and division by zero</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>SEI CERT C++ Coding Standard</th>
<th>VOID INT04-CPP. Enforce limits on integer values originating from untrusted sources</th>
</tr>
</thead>
<tbody>
<tr>
<td>ISO/IEC TS 17961:2013</td>
<td>Forming invalid pointers by library functions ([libptr]) Tainted, potentially mutilated, or out-of-domain integer values are used in a restricted sink ([taintsink])</td>
</tr>
</tbody>
</table>

### Bibliography