THE ART OF SOFTWARE SECURITY ASSESSMENT

Identifying and Avoiding Software Vulnerabilities

MARK DOWD
JOHN McDONALD

THE ART OF SOFTWARE SECURITY ASSESSMENT

Note
In some circumstances, programmers intend to leave the `break` statement out and often leave a comment (such as `/* FALLTHROUGH */`) indicating that the omission of the `break` statement is intentional.

When reviewing code containing `switch` statements, auditors should also determine whether there's any unaccounted-for `case`. Occasionally, `switch` statements lacking a `default case` can be made to effectively run nothing when there isn't a `case` expression matching the result of the expression in the `switch` statement. This error is often an oversight on the developer's part and can lead to unpredictable or undesirable results. Listing 7-24 shows an example.

Listing 7-24
Default Switch Case Omission Vulnerability

```
struct object {
    int type;
    union {
        struct string_obj *str;
        struct int_obj *num;
        struct bool_obj *bool;
    } un;
};

..

struct object *init_object(int type)
{
    struct object *obj;

    if(!(obj = (struct object *)malloc(sizeof(struct object))))
        return NULL;

    obj->type = type;

    switch(type){
        case OBJ_STR:
            obj->un.str = alloc_string();
            break;

        case OBJ_INT:
            obj->un.num = alloc_int();
            break;

        case OBJ_BOOL:
            obj->un.bool = alloc_bool();
            break;
    }

    return obj;
}
```

Listing 7-24 initializes an object based on the supplied `type` variable. The `init_object()` function makes the assumption that the `type` variable supplied is `OBJ_STR`, `OBJ_INT`, or `OBJ_BOOL`. If attackers could supply a value that wasn't any of these three values, this function wouldn't correctly initialize the allocated object structure, which means uninitialized memory could be treated as pointers at a later point in the program.

Auditing Functions

Functions are a ubiquitous component of modern programs, regardless of the application's problem domain or programming language. Application programmers usually divide programs into functions to encapsulate functionality that can be reused in other places in the program and to organize the program into smaller pieces that are easier to conceptualize and manage. Object-oriented programming languages encourage creating member functions, which are organized around objects. As a code auditor, when you encounter a function call, it's important to be cognizant of that call's implications. Ask yourself: What program state changes because of that call? What things can possibly go wrong with that function? What role do arguments play in how that function operates? Naturally, you want to focus on arguments and aspects of the function that users can influence in some way. To formalize this process, look for these four main types of vulnerability patterns that can occur when a function call is made:

- Return values are misinterpreted or ignored.
- Arguments supplied are incorrectly formatted in some way.
- Arguments get updated in an unexpected fashion.
- Some unexpected global program state change occurs because of the function call.

The following sections explore these patterns and explain why they are potentially dangerous.

Function Audit Logs

Because functions are the natural mechanism by which programmers divide their programs into smaller, more manageable pieces, they provide a great way for code auditors to divide their analysis into manageable pieces. This section covers creating an **audit log**, where you can keep notes about locations in the program that could be useful in later analysis. This log is organized around functions and should contain notes on each function's purpose and side effects. Many code auditors use an informal process for keeping these kinds of notes, and the sample audit log used in this section synthesizes some of these informal approaches.

To start, list the basic components of an entry, as shown in Table 7-1, and then you can expand on the log as vulnerabilities related to function interaction are discussed.

Table 7-1

Sample Audit Log

Function prototype	`int read_data(int sockfd, char **buffer, int *length)`
Description	Reads data from the supplied socket and allocates a buffer for storage.
Location	`src/net/read.c`, line 29
Cross-references	`process_request`, `src/net/process.c`, line 400 `process_login`, `src/net/process.c`, line 932
Return value type	32-bit signed integer.
Return value meaning	Indicates error: 0 for success or -1 for error.
Error conditions	`calloc()` failure when allocating `MAX_SIZE` bytes. If read returns less than or equal to 0.
Erroneous return values	When `calloc()` fails, the function returns NULL instead of -1.

While you don't need to understand the entire log yet, the following is a brief summary of each row that you can easily refer back to:

- *Function name*—The complete function prototype.
- *Description*—A brief description of what the function does.
- *Location*—The location of the function definition (file and line number).
- *Cross-references*— The locations that call this function definition (files and line numbers).
- *Return value type*—The C type that is returned.
- *Return value meaning*—The set of return types and the meaning they convey.
- *Error conditions*—Conditions that might cause the function to return error values.
- *Erroneous return values*—Return values that do not accurately represent the functions result, such as not returning an error value when a failure condition occurs.

Return Value Testing and Interpretation

Ignored or misinterpreted return values are the cause of many subtle vulnerabilities in applications. Essentially, each function in an application is a compartmentalized code fragment designed to perform one specific task. Because it does this in a "black box" fashion, details of the results of its operations are largely invisible to calling functions. Return values are used to indicate some sort of status to calling functions. Often this status indicates success or failure or provides a value that's the result of the function's task—whether it's an allocated memory region, an integer result from a mathematical operation, or simply a Boolean true or false to indicate whether a specific operation is allowed. In any case, the return value plays a major part in function calling, in that it communicates some result between two separate functional components. If a return value is misinterpreted or simply ignored, the program might take incorrect code paths as a result, which can have severe security implications. As you can see, a large part of the information in the audit log is related to the return value's meaning and how it's interpreted. The following sections explore the process a code auditor should go through when auditing function calls to determine whether a miscommunication can occur between two components and whether that miscommunication can affect the program's overall security.

Ignoring Return Values

Many functions indicate success or failure through a return value. Consequently, ignoring a return value could cause an error condition to go undetected. In this situation, a code auditor must determine the implications of a function's potential errors going undetected. The following simple example is quite common:

```
char *buf = (char *)malloc(len);

memcpy(buf, src, len);
```

Quite often, the `malloc()` function isn't checked for success or failure, as in the preceding code; the developer makes the assumption that it will succeed. The obvious implication in this example is that the application will crash if `malloc()` can be made to fail, as a failure would cause `buf` to be set to NULL, and the `memcpy()` would cause a NULL pointer dereference. Similarly, it's not uncommon for programmers to fail to check the return value of `realloc()`, as shown in Listing 7-25.

Listing 7-25
Ignoring realloc() Return Value

```
struct databuf
{
    char *data;
    size_t allocated_length;
    size_t used;
};
```

```
...

int append_data(struct databuf *buf, char *src, size_t len)
{
    size_t new_size = buf->used + len + EXTRA;

    if(new_size < len)
        return -1;

    if(new_size > buf->allocated_length)
    {
        buf->data = (char *)realloc(buf->data, new_size);
        buf->allocated_length = new_size;
    }

    memcpy(buf->data + buf->used, src, len);

    buf->used += len;

    return 0;
}
```

As you can see the `buf->data` element can be reallocated, but the `realloc()` return value is never checked for failure. When the subsequent `memcpy()` is performed, there's a chance an exploitable memory corruption could occur. Why? Unlike the previous `malloc()` example, this code copies to an offset from the allocated buffer. If `realloc()` fails, `buf->data` is NULL, but the `buf->used` value added to it might be large enough to reach a valid writeable page in memory.

Ignoring more subtle failures that don't cause an immediate crash can lead to far more serious consequences. Paul Starzetz, an accomplished researcher, discovered a perfect example of a subtle failure in the Linux kernel's memory management code. The `do_mremap()` code is shown in Listing 7-26.

Listing 7-26

Linux do_mremap() Vulnerability

```
        /* new_addr is valid only if MREMAP_FIXED is
           specified */
        if (flags & MREMAP_FIXED) {
                if (new_addr & ~PAGE_MASK)
                        goto out;
                if (!(flags & MREMAP_MAYMOVE))
                        goto out;

                if (new_len > TASK_SIZE
                    || new_addr > TASK_SIZE - new_len)
                        goto out;
                /* Check if the location you're moving into
                 * overlaps the old location at all, and
                 * fail if it does.
                 */
                if ((new_addr <= addr)
                    && (new_addr+new_len) > addr)
                        goto out;

                if ((addr <= new_addr)
                    && (addr+old_len) > new_addr)
                        goto out;

                do_munmap(current->mm, new_addr, new_len);
        }

        /*
         * Always allow a shrinking remap: that just unmaps
         * the unnecessary pages.
         */
        ret = addr;
        if (old_len >= new_len) {
                do_munmap(current->mm, addr+new_len,
                          old_len - new_len);
                if (!(flags & MREMAP_FIXED)
                    || (new_addr == addr))
                        goto out;
        }
```

The vulnerability in this code is that the `do_munmap()` function could be made to fail. A number of conditions can cause it to fail; the easiest is exhausting maximum resource limits when splitting an existing virtual memory area. If the `do_munmap()` function fails, it returns an error code, which `do_mremap()` completely ignores. The result of ignoring this return value is that the virtual memory area (VMA) structures used to represent page ranges for processes can be made inconsistent by having page table entries overlapped in two VMAs or totally unaccounted-for VMAs. Through a novel exploitation method using the page-caching system, arbitrary pages could be mapped erroneously into other processes, resulting in a privilege escalation condition. More information on this vulnerability is available at www.isec.pl/vulnerabilities/isec-0014-mremap-unmap.txt.

Generally speaking, if a function call returns a value, as opposed to returning nothing (such as a void function), a conditional statement should follow each function call to test for success or failure. Notable exceptions are cases in which the function terminates the application via a call to an exit routine or errors are handled by an exception mechanism in a separate block of code. If no check is made to test for success or failure of a function, the code auditor should take note of the location where the value is untested.