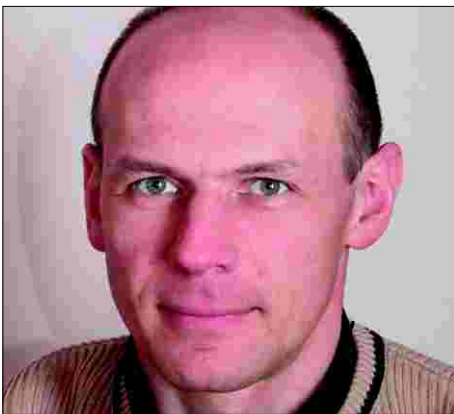


Beware of buffer overflows!

Tightening military & aerospace security through unit testing

Security vulnerabilities in software for military and aerospace systems can be far more dangerous than the functional problems, for which industry has developed so many control tools. For instance an error that affects navigation or fuel could have disastrous consequences.



Sergei Sokolov
of Parasoft Corporation.

About the author

Sergei Sokolov is a manager of professional services at Parasoft Corporation. Prior to that, he developed electronic design automation software at ASC, Sente, and Sequence Design.



When an attacker manages to exploit security vulnerability in the software, the resulting unexpected behavior is potentially the start of a crisis. After all, someone attacking a system is typically set on being as malicious as possible. How can we prepare for such an onslaught? In fact, many of the same techniques that can be used to prevent functional problems can also be used to reduce security vulnerabilities. This article looks at buffer overflows, the most common type of attack used to exploit military and aerospace systems, and explains how best practices such as unit testing and coverage analysis can help prevent these attacks.

Beware of buffer overflows

Buffer overflows are becoming far too much of a standard problem affecting C and C++ applications. Buffer overflow attacks occur when a hacker manages to break past an application's perimeter security, pass an input through the programme's built-in defences, and write to the buffer. These attacks can only occur when the attacker is able to find and exploit a memory corruption bug in the application.

For example, assume that your C++ application has an array or a memory chunk on the stack, and a memory issue makes it possible to write beyond the array or memory chunk, and overwrite the return address of the function. The hacker can exploit this weakness so that the function returns to a hacker-designated function, or so that the function executes a hacker-

designated operation. In fact, some recently-discovered buffer overflows in major operating systems give hackers licence to perform a variety of malicious actions, such as causing the system to fail, installing programs, viewing, changing, and deleting data, modifying any part of the system, and creating accounts with full privileges.

Developers traditionally try to handle these exploits by limiting the size of the input or by verifying the input. However, it is easy to miss cases if you have no procedure for identifying all the inputs that need to be checked. To actually remove the opportunity for these attacks, you need to prevent the memory corruption bugs that allow them to occur.

Most dangerous: buffer overflows that change a function's return address.

The most dangerous type of buffer overflow is typically a buffer overflow of an array that is allocated on the stack. If it is possible to write beyond an array, it is also possible to overwrite the return address of a function; if this occurs, the function can return to a place that the developer did not intend. If an attacker is trying to manipulate the system, he could exploit this weakness by providing inputs that overwrite the function's return address, have the function return to a place of his choice instead, and then perform operations of his choice.

This type of buffer overflow can be prevented with unit testing. To use unit testing to expose potential vulnerabilities, you take every routine that allocates arrays on the stack and try to generate a wide range and scope of test cases for each array. The goal is to determine whether any potential arguments can overwrite memory on the stack. These test cases can be generated automatically with a unit testing tool and/or created manually. To determine whether memory corruption occurs as these test cases execute, you run the unit

```

static const int stPacketArraySize = 1024;
...
ErrorStatus processUserDataStream(int userStreamLength, UserStream& stream)
{
    ErrorStatus status = OK;
    Packet packetArray[stPacketArraySize];
    if (userStreamLength >= stPacketArraySize) // (*)
    {
        status = BUFFER_TOO_SMALL;
    }
    else
    {
        Packet* packet = processPacket(stream);
        for (int i = 0; packet; i++)
        {
            packetArray[i] = packet;
            packet = processPacket();
        }
        // further process the packetArray
        ...
        return status;
    }
}

```

Fig. 1

tests under a runtime error detection tool that provides memory access checking.

For example, consider Figure 1.

Although this function has a legitimate check on the array size, the check (*) will not trigger if the array size does not correspond to the actual stream length or is negative (e.g. 1), and a buffer overflow may result. This situation is a cause for concern, especially if the stream length is provided by the user as a part of the stream. Using unit testing, we could identify this critical security vulnerability. For example, testing the function with the typical set of boundary conditions for an integer variable (-MAX_INT, -1, 0, 1, MAX_INT) and adding another test condition when the supplied stream length parameter is shorter than the actual would expose this problem.

Once the error is found, an appropriate fix would require not only checking the size of the array against the stream length upfront, but also the array index in the loop where the processed packets are stored: Figure 2.

By leveraging unit testing for security verification purposes, we identified and removed this flaw as soon as the unit was completed and prevented the security vulnerability from ever reaching the integrated application. As study after study shows, the earlier a flaw is found, the easier, faster, and less expensive it is to correct.

Memory overwrites that change execution paths

Even if an attacker cannot overwrite memory on the stack, he can still perform serious damage by altering the expected execution path. Any overwritten memory could be dangerous if that memory affects the direction of the program execution. For instance, you have code with an **if/else** branch statement and two arrays. The values in the second array influence the branch taken.

Using username of the maximum field length will result in a memory overwrite by

```

for (int i = 0; i < stPacketArraySize && packet; i++)
{
    packetArray[i] = packet;
    packet = processPacket();
}

```

Fig. 2

```

AccessCode RequestAccess(ResourceID resource, const char* username)
{
    AccessCode permission;
    char nameBuf[USERNAME_FIELD_LENGTH];
    bool accessFlags[NUMBER_OF_FLAGS];
    // for a bogus username the access flags should be set to "deny"
    RetrieveAccessFlags(username, resource, accessFlags);
    // nameBuf buffer may overflow and spill into the accessFlags
    // array, overwriting the data there and possibly allowing access
    // to the requested resource
    strcpy(nameBuf, username);
    if (accessFlags[0] && accessFlags[2])
    {
        permission = DENY_ACCESS;
        DenyAccessMessage(nameBuf);
    }
    else if (accessFlags[3])
    {
        permission = REQUIRE_CONFIRMATION;
        RequireConfirmationMessage(nameBuf);
    }
    else
        permission = GRANT_ACCESS;
}

```

Fig. 3

strcpy, as it will need to copy a terminating 0 into the **nameBuf** as well. Thus, this memory bug allows an attacker to alter the branch selection that the program execution will normally take. Assuming that the branch selection statement controls the assignment of security privileges, an attacker could exploit this vulnerability to upgrade his security privileges without even inserting any special instructions into the exploit string passed as a username.

To determine whether it is possible to overwrite memory in this code, you create unit tests, then run the unit tests under a runtime error detection tool that provides memory access checking. The goal is to exercise the first array as thoroughly as possible to determine whether an attacker can possibly write beyond that array.

For instance, the following simple test case would expose the security vulnerability in the previous piece of code: Figure 3. Upon catching the memory error, the first fix is to properly size **nameBuf** (increasing it by 1). Then, the rules on checking on the length of the string being copied should apply. Preferably, the **nameBuf** buffer should be sized by using **strlen(username)**: Figure 4.

Vital considerations

When applying unit testing to expose security vulnerabilities, it's important to remember that if unit tests do not thoroughly exercise the unit, memory corruption may go unnoticed, opening the door to security attacks. That's why it is critical to monitor the coverage of your tests. When performing unit testing in an attempt to expose security vulnerabilities, a goal of 100% coverage of each function is both desirable and achievable. This degree of coverage is possible at the unit level because it is so much easier to design inputs that reach all parts of the function when you test a function in isolation (apart from the rest of the application) than it is when you test at the application level.

Another important point to consider is that every instance of memory corruption should be seen as a potential security vulnerability. In many cases, human subjectivity is actually more of a hindrance than a help in evaluating potential security vulnerabilities. For instance, let's return to the example in the previous section. This time, assume that the switch statement direction is controlled by values in an array, but you do not expect memory to be laid out in such a way that this critical array follows the first array. If you had these expectations and you learned that the first array might overwrite memory, you would probably assume that this overwrite would not affect the critical array, and you might decide against correcting the memory corruption. However, this might be a bad decision. If your expectations about memory layout are not met, the code could indeed be vulnerable to security attacks. And your memory layout expectations might not be met because it is difficult to predict how compilers will lay out memory. Different compilers lay out memory in different ways. If you can't be certain how memory will be laid out, you can't be certain about how writing beyond the bounds of an array will impact the application. Thanks to the variation among compilers, the exact same code that appears to be secure when compiled on one compiler could be vulnerable to attack when compiled on another compiler.

There are two key lessons to be learned

here. One, because memory layout is compiler-dependent, you cannot accurately identify potential memory corruption and the security vulnerabilities they bring by simply examining or parsing the source code; it must be compiled, executed, and monitored as it executes. Two, you can never be too thorough or too paranoid when it comes to identifying and removing the possibility for memory corruption. With software designed for use in the military and aerospace systems, a security attack can have severe and potentially deadly consequences.

To reduce the risk of attackers gaining unauthorized control over mission-critical systems, it is crucial that you identify and correct every potential memory corruption in the system software ■

ssokolov@parasoft.com

Subscribe to
Professional Tester Magazine

```
const char* username = "username of at least the input
field length";
```

Fig. 4

```
unsigned int bufLength = strlen(username) + 1;
char nameBuf[bufLength];
```

Fig. 5

IN-HOUSE COURSES ON RISK-BASED TESTING AND PROJECT MANAGEMENT

by Felix Redmill

Improve effectiveness by
applying risk-based thinking

Heard of riskbased
testing?
Ever met anyone
who
understands it?
Felix Redmill
brings an
understanding
of risk from the
safetycritical
systems
domain. He teaches
risk principles and
their use in
improving test
effectiveness and
project
management.

Risk-based Testing
in Practice

2-day course for test
planners,
managers and testers

The Benefits of
Risk-based Testing

Half-day appreciation for
managers

Managing Project Risks

1.5-day course for project
personnel

Redmill Consultancy

22 Onslow Gardens
London N10 3JU, UK

Tel: +44 (0)20 8883 0789

Email: Felix.Redmill@ncl.ac.uk