Fortify Security Report

Sep 28, 2017

ChandlerAustin

Executive Summary

Issues Overview

On Sep 28, 2017, a source code review was performed over the stackbuffer-c code base. 8 files, 7 LOC (Executable) were scanned and reviewed for defects that could lead to potential security vulnerabilities. A total of 2 reviewed findings were uncovered during the analysis.

Issues by Fortify Priority Order

Critical	1
Low (1 Hidden)	1

Recommendations and Conclusions

The Issues Category section provides Fortify recommendations for addressing issues at a generic level. The recommendations for specific fixes can be extrapolated from those generic recommendations by the development group.

Project Summary

Code Base Summary

Code location: C:/Users/ChandlerAustin/Documents/lab3/task6
Number of Files: 8
Lines of Code: 7
Build Label: <No Build Label>

Scan Information

Scan time: 00:13
SCA Engine version: 6.21.0007
Machine Name: IALAB04
Username running scan: ChandlerAustin

Results Certification

Results Certification Valid

Details:

Results Signature:

SCA Analysis Results has Valid signature

Rules Signature:

There were no custom rules used in this scan

Attack Surface

Attack Surface:

Filter Set Summary

Current Enabled Filter Set:
Quick View

Filter Set Details:

Folder Filters:
If [fortify priority order] contains critical Then set folder to Critical
If [fortify priority order] contains high Then set folder to High
If [fortify priority order] contains medium Then set folder to Medium
If [fortify priority order] contains low Then set folder to Low
Visibility Filters:
If impact is not in range [2.5, 5.0] Then hide issue
If likelihood is not in range (1.0, 5.0] Then hide issue

Audit Guide Summary

Audit guide not enabled

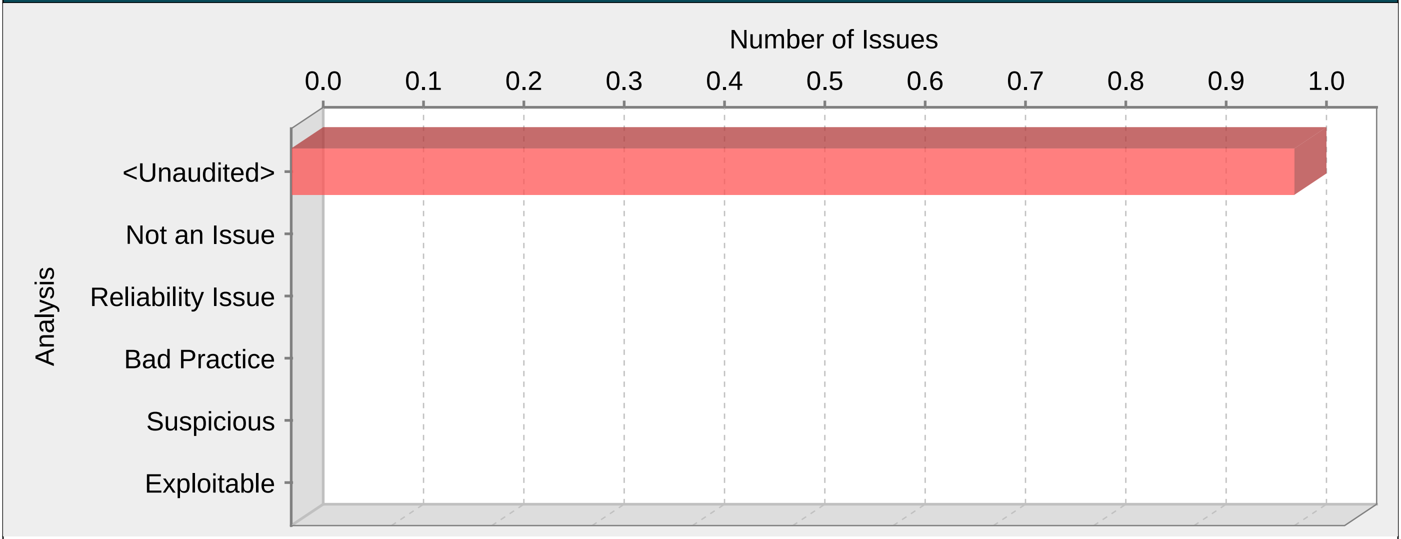
Results Outline

Overall number of results

The scan found 2 issues.

Vulnerability Examples by Category

Category: Buffer Overflow (1 Issues)



Abstract:

Writing outside the bounds of a block of allocated memory can corrupt data, crash the program, or cause the execution of malicious code.

Explanation:

Buffer overflow is probably the best known form of software security vulnerability. Most software developers know what a buffer overflow vulnerability is, but buffer overflow attacks against both legacy and newly-developed applications are still quite common. Part of the problem is due to the wide variety of ways buffer overflows can occur, and part is due to the error-prone techniques often used to prevent them.

In a classic buffer overflow exploit, the attacker sends data to a program, which it stores in an undersized stack buffer. The result is that information on the call stack is overwritten, including the function's return pointer. The data sets the value of the return pointer so that when the function returns, it transfers control to malicious code contained in the attacker's data.

Although this type of stack buffer overflow is still common on some platforms and in some development communities, there are a variety of other types of buffer overflow, including heap buffer overflows and off-by-one errors among others. There are a number of excellent books that provide detailed information on how buffer overflow attacks work, including Building Secure Software [1], Writing Secure Code [2], and The Shellcoder's Handbook [3].

At the code level, buffer overflow vulnerabilities usually involve the violation of a programmer's assumptions. Many memory manipulation functions in C and C++ do not perform bounds checking and can easily overwrite the allocated bounds of the buffers they operate upon. Even bounded functions, such as strncpy(), can cause vulnerabilities when used incorrectly. The combination of memory manipulation and mistaken assumptions about the size or makeup of a piece of data is the root cause of most buffer overflows.

Buffer overflow vulnerabilities typically occur in code that:

- Relies on external data to control its behavior.

- Depends upon properties of the data that are enforced outside of the immediate scope of the code.

- Is so complex that a programmer cannot accurately predict its behavior.

The following examples demonstrate all three of the scenarios.

Example 1: This is an example of the second scenario in which the code depends on properties of the data that are not verified locally. In this example a function named lccopy() takes a string as its argument and returns a heap-allocated copy of the string with all uppercase letters converted to lowercase. The function performs no bounds checking on its input because it expects str to always be smaller than BUFSIZE. If an attacker bypasses checks in the code that calls lccopy(), or if a change in that code makes the assumption about the size of str untrue, then lccopy() will overflow buf with the unbounded call to strcpy().

```
char *lccopy(const char *str) {
char buf[BUFSIZE];
char *p;
strcpy(buf, str);
for (p = buf; *p; p++) {
if (isupper(*p)) {
*p = tolower(*p);
}
}
return strdup(buf);
}
```

Example 2.a: The following sample code demonstrates a simple buffer overflow that is often caused by the first scenario in which the code relies on external data to control its behavior. The code uses the gets() function to read an arbitrary amount of data into a stack buffer. Because there is no way to limit the amount of data read by this function, the safety of the code depends on the user to always enter fewer than BUFSIZE characters.

```
...
char buf[BUFSIZE];
gets(buf);
...
```

Example 2.b: This example shows how easy it is to mimic the unsafe behavior of the gets() function in C++ by using the >> operator to read input into a char[] string.

```
...
char buf[BUFSIZE];
cin >> (buf);
...
```

Example 3: The code in this example also relies on user input to control its behavior, but it adds a level of indirection with the use of the bounded memory copy function memcpy(). This function accepts a destination buffer, a source buffer, and the number of bytes to copy. The input buffer is filled by a bounded call to read(), but the user specifies the number of bytes that memcpy() copies.

```
...
char buf[64], in[MAX_SIZE];
printf("Enter buffer contents:\n");
read(0, in, MAX_SIZE-1);
printf("Bytes to copy:\n");
scanf("%d", &bytes);
memcpy(buf, in, bytes);
...
```

Note: This type of buffer overflow vulnerability (where a program reads data and then trusts a value from the data in subsequent memory operations on the remaining data) has turned up with some frequency in image, audio, and other file processing libraries.

Example 4: The following code demonstrates the third scenario in which the code is so complex its behavior cannot be easily predicted. This code is from the popular libPNG image decoder, which is used by a wide array of applications, including Mozilla and some versions of Internet Explorer.

The code appears to safely perform bounds checking because it checks the size of the variable length, which it later uses to control the amount of data copied by png_crc_read(). However, immediately before it tests length, the code performs a check on png_ptr->mode, and if this check fails a warning is issued and processing continues. Because length is tested in an else if block, length would not be tested if the first check fails, and is used blindly in the call to png_crc_read(), potentially allowing a stack buffer overflow.

Although the code in this example is not the most complex we have seen, it demonstrates why complexity should be minimized in code that performs memory operations.

```
if (!(png_ptr->mode & PNG_HAVE_PLTE)) {
/* Should be an error, but we can cope with it */
png_warning(png_ptr, "Missing PLTE before tRNS");
}
else if (length > (png_uint_32)png_ptr->num_palette) {
png_warning(png_ptr, "Incorrect tRNS chunk length");
png_crc_finish(png_ptr, length);
return;
}
...
png_crc_read(png_ptr, readbuf, (png_size_t)length);
```

Example 5: This example also demonstrates the third scenario in which the program's complexity exposes it to buffer overflows. In this case, the exposure is due to the ambiguous interface of one of the functions rather than the structure of the code (as was the case in the previous example).

The getUserInfo() function takes a username specified as a multibyte string and a pointer to a structure for user information, and populates the structure with information about the user. Since Windows authentication uses Unicode for usernames, the username argument is first converted from a multibyte string to a Unicode string. This function then incorrectly passes the size of unicodeUser in bytes rather than characters. The call to MultiByteToWideChar() may therefore write up to (UNLEN+1)*sizeof(WCHAR) wide characters, or

(UNLEN+1)*sizeof(WCHAR)*sizeof(WCHAR) bytes, to the unicodeUser array, which has only (UNLEN+1)*sizeof(WCHAR) bytes allocated. If the username string contains more than UNLEN characters, the call to MultiByteToWideChar() will overflow the buffer unicodeUser.

```
void getUserInfo(char *username, struct _USER_INFO_2 info){
WCHAR unicodeUser[UNLEN+1];
MultiByteToWideChar(CP_ACP, 0, username, -1,
unicodeUser, sizeof(unicodeUser));
NetUserGetInfo(NULL, unicodeUser, 2, (LPBYTE *)&info);
}
```

Recommendations:

Never use inherently unsafe functions, such as gets(), and avoid the use of functions that are difficult to use safely such as strcpy(). Replace unbounded functions like strcpy() with their bounded equivalents, such as strncpy() or the WinAPI functions defined in strsafe.h [4].

Although the careful use of bounded functions can greatly reduce the risk of buffer overflow, this migration cannot be done blindly and does not go far enough on its own to ensure security. Whenever you manipulate memory, especially strings, remember that buffer overflow vulnerabilities typically occur in code that:

- Relies on external data to control its behavior

- Depends upon properties of the data that are enforced outside of the immediate scope of the code

- Is so complex that a programmer cannot accurately predict its behavior.

Additionally, consider the following principles:

- Never trust an external source to provide correct control information to a memory operation.

- Never trust that properties about the data your program is manipulating will be maintained throughout the program. Sanity check data before you operate on it.

- Limit the complexity of memory manipulation and bounds-checking code. Keep it simple and clearly document the checks you perform, the assumptions that you test, and what the expected behavior of the program is in the case that input validation fails.

- When input data is too large, be leery of truncating the data and continuing to process it. Truncation can change the meaning of the input.

- Do not rely on tools, such as StackGuard, or non-executable stacks to prevent buffer overflow vulnerabilities. These approaches do not address heap buffer overflows and the more subtle stack overflows that can change the contents of variables that control the program. Additionally, many of these approaches are easily defeated, and even when they are working properly, they address the symptom of the problem and not its cause.

Tips:

1. On Windows, less secure functions like memcpy() can be replaced with their more secure versions, such as memcpy_s(). However, this still needs to be done with caution. Because parameter validation provided by the _s family of functions varies, relying on it can lead to unexpected behavior. Furthermore, incorrectly specifying the size of the destination buffer can still result in buffer overflows.

stackbuffer.c, line 13 (Buffer Overflow)

Fortify Priority:	Critical	**Folder**	Critical
Kingdom:	Input Validation and Representation		
Abstract:	The function doMemCpy() in stackbuffer.c might be able to write outside the bounds of allocated memory on line 13, which could corrupt data, cause the program to crash, or lead to the execution of malicious code.		
Source:	stackbuffer.c:24 scanf()		

```
22    read(0, in, MAX_SIZE-1);
23    printf("Bytes to copy:\n");
24    scanf("%d", &bytes);
25
26    doMemCpy(buf, in, bytes);
```

Sink: stackbuffer.c:13 memcpy()

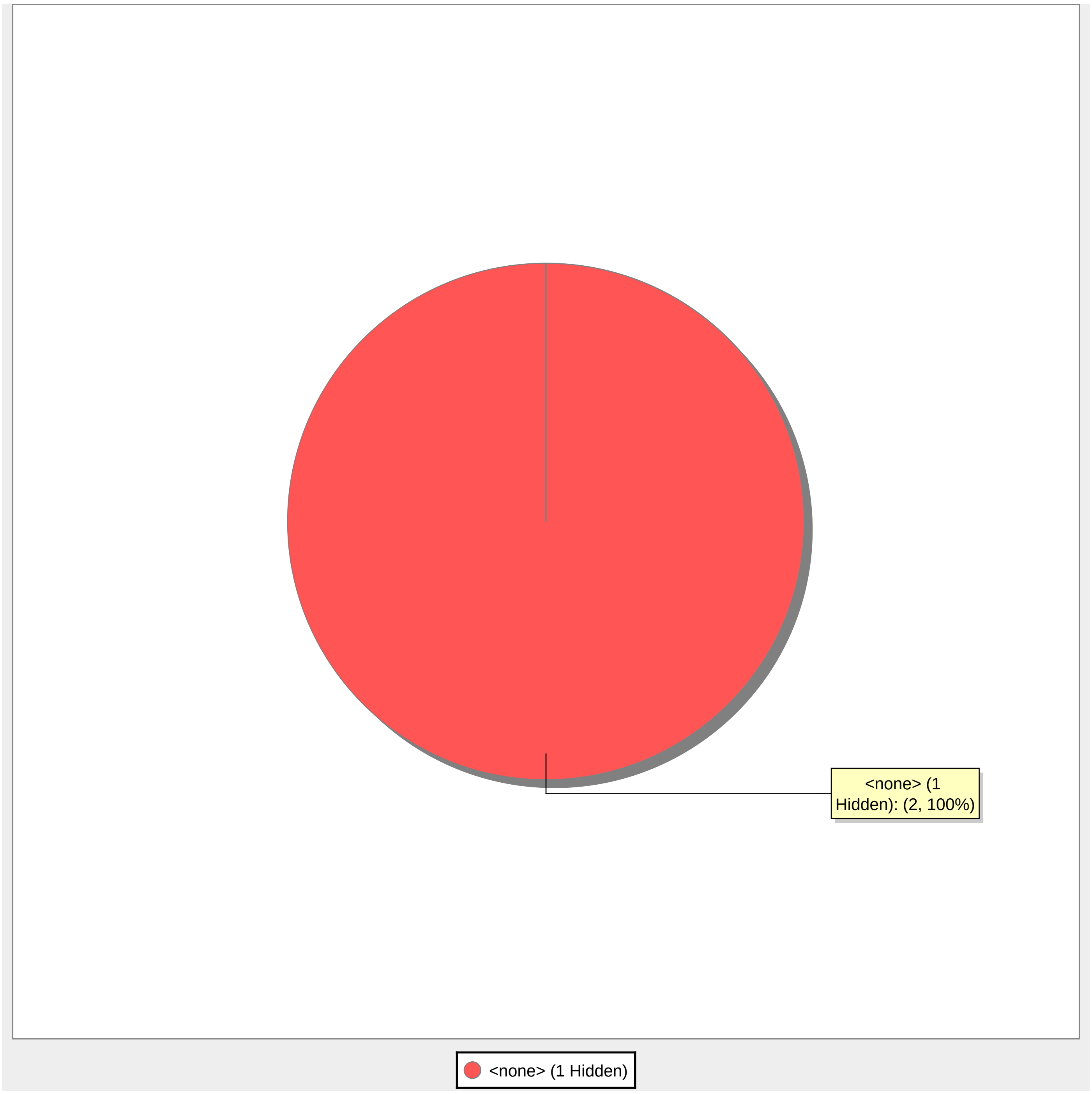
```
11
12  void doMemCpy(char* buf, char* in, int chars) {
13    memcpy(buf, in, chars);
14  }
```

Issue Count by Category

Issues by Category	
Buffer Overflow	1
Unchecked Return Value (1 Hidden)	1

Issue Breakdown by Analysis

Issues by Analysis


<none> (1 Hidden): (2, 100%)
<none> (1 Hidden)