Fortify Security Report

Sep 28, 2017

ChandlerAustin

Executive Summary

Issues Overview

On Sep 28, 2017, a source code review was performed over the sample-cpp code base. 78 files, 30 LOC (Executable) were scanned and reviewed for defects that could lead to potential security vulnerabilities. A total of 4 reviewed findings were uncovered during the analysis.

Issues by Fortify Priority Order

Critical	2
High (2 Hidden)	2

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/task1
Number of Files: 78
Lines of Code: 30
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

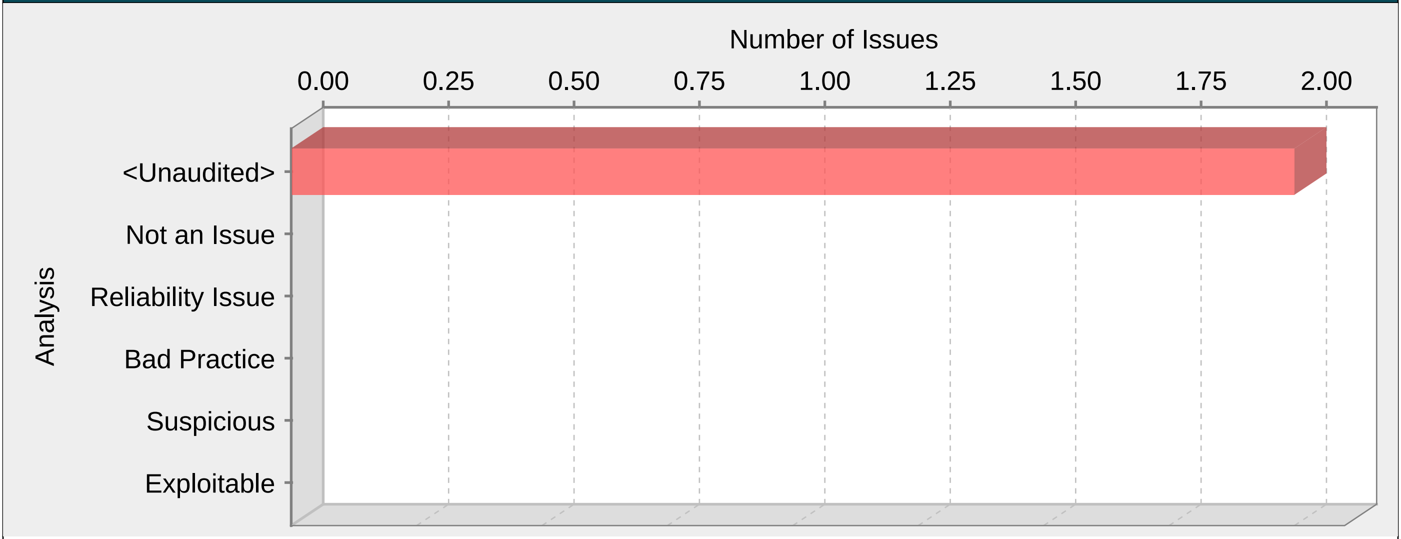
Results Outline

Overall number of results

The scan found 4 issues.

Vulnerability Examples by Category

Category: Command Injection (2 Issues)



Abstract:

Executing commands that include unvalidated user input can cause an application to act on behalf of an attacker.

Explanation:

Command injection vulnerabilities take two forms:

- An attacker can change the command that the program executes: the attacker explicitly controls what the command is.

- An attacker can change the environment in which the command executes: the attacker implicitly controls what the command means.

In this case we are primarily concerned with the first scenario, in which an attacker explicitly controls the command that is executed. Command injection vulnerabilities of this type occur when:

1. Data enters the application from an untrusted source.

2. The data is part of a string that is executed as a command by the application.

3. By executing the command, the application gives an attacker a privilege or capability that the attacker would not otherwise have.

Example 1: The following simple program accepts a filename as a command line argument and displays the contents of the file back to the user. The program is installed setuid root because it is intended for use as a learning tool to allow system administrators in-training to inspect privileged system files without giving them the ability to modify them or damage the system.

```
int main(char* argc, char** argv) {
char cmd[CMD_MAX] = "/usr/bin/cat ";
strcat(cmd, argv[1]);
system(cmd);
}
```

Because the program runs with root privileges, the call to system() also executes with root privileges. If a user specifies a standard filename, the call works as expected. However, if an attacker passes a string of the form ";rm -rf /", then the call to system() fails to execute cat due to a lack of arguments and then plows on to recursively delete the contents of the root partition.

Example 2: The following code from a privileged program uses the environment variable $APPHOME to determine the application's installation directory and then executes an initialization script in that directory.

```
...
char* home=getenv("APPHOME");
char* cmd=(char*)malloc(strlen(home)+strlen(INITCMD));
if (cmd) {
```

```
strcpy(cmd,home);
strcat(cmd,INITCMD);
execl(cmd, NULL);
}
...
```

As in Example 1, the code in this example allows an attacker to execute arbitrary commands with the elevated privilege of the application. In this example, the attacker can modify the environment variable $APPHOME to specify a different path containing a malicious version of INITCMD. Because the program does not validate the value read from the environment, by controlling the environment variable the attacker can fool the application into running malicious code.

The attacker is using the environment variable to control the command that the program invokes, so the effect of the environment is explicit in this example. We will now turn our attention to what can happen when the attacker can change the way the command is interpreted.

Example 3: The code below is from a web-based CGI utility that allows users to change their passwords. The password update process under NIS includes running make in the /var/yp directory. Note that since the program updates password records, it has been installed setuid root.

The program invokes make as follows:

```
system("cd /var/yp && make &> /dev/null");
```

Unlike the previous examples, the command in this example is hardcoded, so an attacker cannot control the argument passed to system(). However, since the program does not specify an absolute path for make and does not scrub any environment variables prior to invoking the command, the attacker can modify their $PATH variable to point to a malicious binary named make and execute the CGI script from a shell prompt. And since the program has been installed setuid root, the attacker's version of make now runs with root privileges.

On Windows, additional risks are present.

Example 4: When invoking CreateProcess() either directly or via a call to one of the functions in the _spawn() family, care must be taken when there is a space in an executable or path.

```
...
LPTSTR cmdLine = _tcsdup(TEXT("C:\\Program Files\\MyApplication -L -S"));
CreateProcess(NULL, cmdLine, ...);
...
```

Because of the way CreateProcess() parses spaces, the first executable the operating system will try to execute is Program.exe, not MyApplication.exe. Therefore, if an attacker is able to install a malicious application called Program.exe on the system, any program that incorrectly calls CreateProcess() using the Program Files directory will run this application instead of the intended one.

The environment plays a powerful role in the execution of system commands within programs. Functions like system(), exec(), and CreateProcess() use the environment of the program that calls them, and therefore attackers have a potential opportunity to influence the behavior of these calls.

Recommendations:

Do not allow users to have direct control over the commands executed by the program. If user input affects the command to be run, use the input only to select from a predetermined set of safe commands. If the input appears to be malicious, the value passed to the command execution function should either default to some safe selection from this set or the program should decline to execute any command.

If user input must be used as an argument to a command executed by the program, this solution can become impractical; the set of legitimate argument values may be too large or too hard to keep track of. In this situation, programmers often fall back on blacklisting to selectively reject or escape potentially dangerous characters before using the input. However, any such list of unsafe characters is likely to be incomplete and will be heavily dependent on the system where the commands are executed. A better approach is to create a whitelist of characters that are allowed to appear in the input and accept input composed exclusively of characters in the approved set.

Another line of defense against maliciously crafted input is to avoid the use of functions that perform shell interpretation. For example, do not use system(), which executes its own command shell.

Be aware of the external environment and how it affects the behavior of the commands you execute. In particular, pay attention to how the $PATH, $LD_LIBRARY_PATH, and $IFS variables are used on Unix and Linux machines.

Be aware that the Windows APIs impose a specific search order that is based not only on a series of directories, but also on a list of file extensions that are automatically appended if none is specified. For example, functions in the _spawn() family, try executing file name extensions in the following order, if the command name argument does not have a file name extension or does not end with a period: first .com, then .exe, then .bat, and finally .cmd. Furthermore, additional risks exist on Windows due to the way command executing functions parse spaces in arguments that represent executables and paths.

Example 5: The code below re-writes Example 4 to avoid unintentionally executing a malicious application by using quotation marks around the executable path.

```
...
LPTSTR cmdLine[] = _tcsdup(TEXT("\"C:\\Program Files\\MyApplication\" -L -S"));
CreateProcess(NULL, cmdLine, ...);
...
```

Another way to achieve the same result is to pass the name of the executable as the first argument, instead of passing NULL.

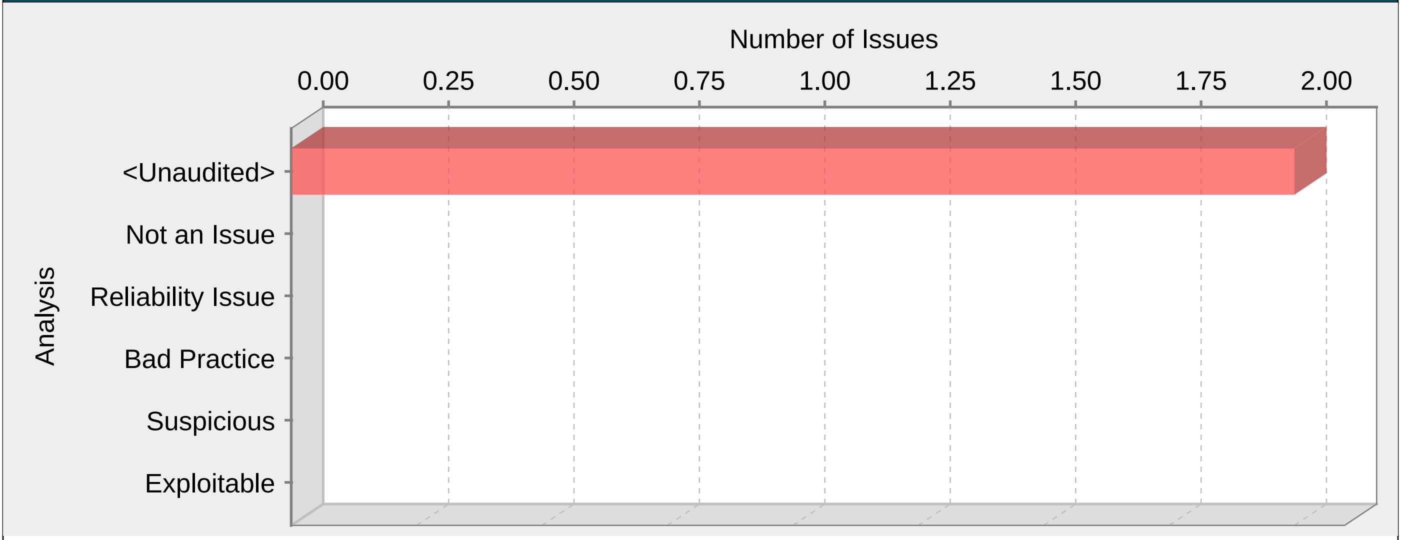
Although it may be impossible to completely protect a program from an imaginative attacker bent on controlling the commands the program executes, be sure to apply the principle of least privilege wherever the program executes an external command: do not hold privileges that are not essential to the execution of the command.

sample.cpp, line 22 (Command Injection)

Fortify Priority:	Critical	**Folder**	Critical
Kingdom:	Input Validation and Representation		
Abstract:	The function Shell() in sample.cpp calls system() on line 22 to execute a command built from untrusted data. This allows an attacker to inject malicious commands.		
Source:	istream:428 std::basic_istream::getline()		
Sink:	sample.cpp:22 system()		

```
20          break;
21        }
22        returnCode = system(cmd);
23        cout << "Command returned " << returnCode << '\n';
24      }
```

Category: Memory Leak (2 Issues: 2 Hidden)



Abstract:

Memory is allocated but never freed.

Explanation:

Memory leaks have two common and sometimes overlapping causes:

- Error conditions and other exceptional circumstances.

- Confusion over which part of the program is responsible for freeing the memory.

Most memory leaks result in general software reliability problems, but if an attacker can intentionally trigger a memory leak, the attacker may be able to launch a denial of service attack (by crashing the program) or take advantage of other unexpected program behavior resulting from a low memory condition [1].

Example 1: The following C function leaks a block of allocated memory if the call to read() fails to return the expected number of bytes:

```
char* getBlock(int fd) {
char* buf = (char*) malloc(BLOCK_SIZE);
if (!buf) {
return NULL;
}
if (read(fd, buf, BLOCK_SIZE) != BLOCK_SIZE) {
return NULL;
}
return buf;
}
```

Recommendations:

Because memory leaks can be difficult to track down, you should establish a set of memory management patterns and idioms for your software. Do not tolerate deviations from your conventions.

One good pattern for addressing the error handling mistake in the example is to use forward-reaching goto statements so that the function has a single well-defined region for handling errors, as follows:

```
char* getBlock(int fd) {
char* buf = (char*) malloc(BLOCK_SIZE);
if (!buf) {
goto ERR;
}
if (read(fd, buf, BLOCK_SIZE) != BLOCK_SIZE) {
goto ERR;
}
return buf;

ERR:
if (buf) {
```

free(buf);
}
return NULL;
}

Tips:

1. Managed pointer objects, such as C++ auto_ptr and Boost smart pointers, are used to ensure that referenced memory allocations are freed. However, memory leaks can still occur when auto_ptrs or certain types of Boost smart pointers are used to reference arrays, Boost array pointers reference individual objects, and the auto_ptr release method is used.

sample.cpp, line 12 (Memory Leak) [Hidden]

Fortify Priority:	High	**Folder**	High
Kingdom:	Code Quality		

Abstract: The function Shell() in sample.cpp allocates memory on line 12 and fails to free it.

Sink: sample.cpp:12 cmd = operator new[](...)

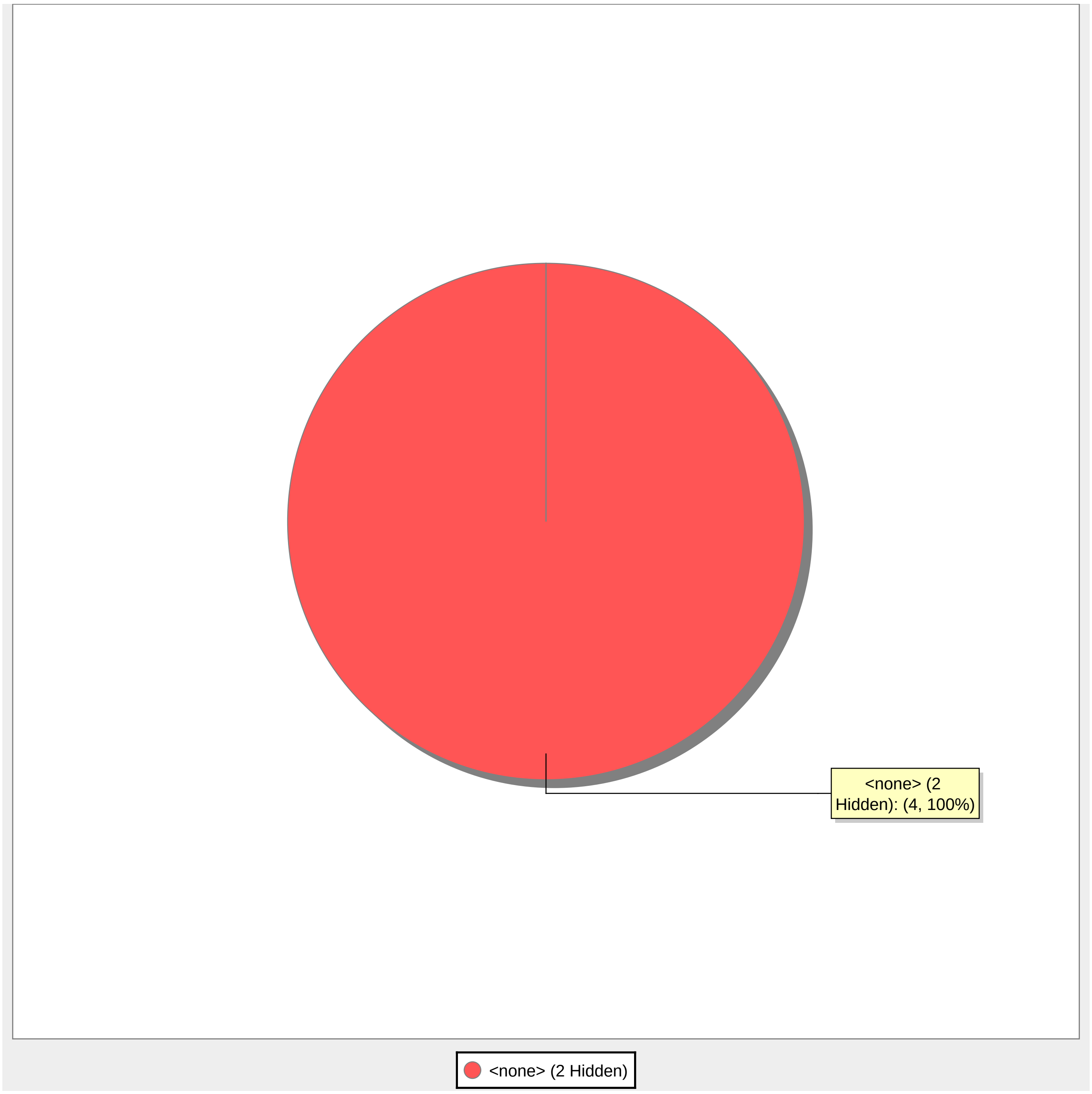
```
10
11                     Shell() {
12                      char *cmd = new char[256];
13                      const char *safe = "safe_program ";
14                      int returnCode;
```

Issue Count by Category

Issues by Category	
Command Injection	2
Memory Leak (2 Hidden)	2

Issue Breakdown by Analysis

Issues by Analysis


<none> (2 Hidden): (4, 100%)
● <none> (2 Hidden)