Fortify Developer Workbook

2017-11-21

Report Overview

Report Summary

On 2017-11-21, a source code review was performed over the getlog code base. 59 files, 500 LOC (Executable) were scanned. A total of 25 issues were uncovered during the analysis. This report provides a comprehensive description of all the types of issues found in this project. Specific examples and source code are provided for each issue type.

Issues by Fortify Priority Order	
Low	16
Critical	6
High	3

Issue Summary

Overall number of results

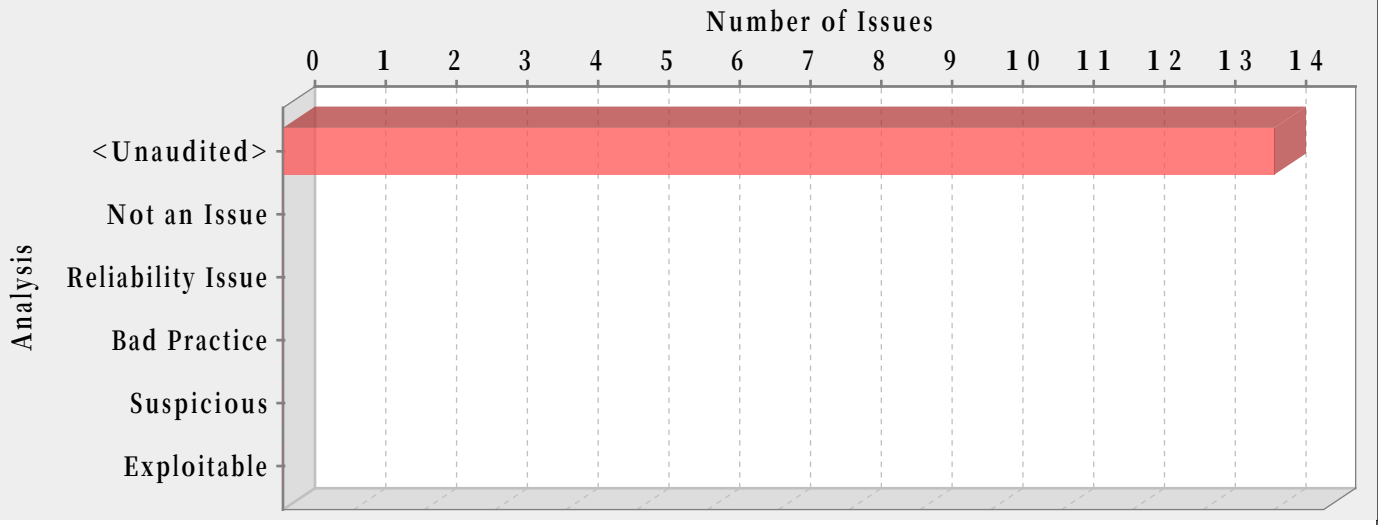
The scan found 25 issues.

Issues by Category	
Dangerous Function: strcpy()	14
Buffer Overflow	7
Path Manipulation	2
Command Injection	1
System Information Leak: Internal	1

Results Outline

Vulnerability Examples by Category

Category: Dangerous Function: strcpy() (14 Issues)



Abstract:
无法安全地使用函数 strcpy()。不应该使用此函数。

Explanation:
某些函数不论如何使用都有危险性。这一类函数通常是在没有考虑安全问题的情况下就执行了。

strcpy() 函数不安全，因为它假设其输入以 "\0" 结尾，并且分配了足够的内存，以便在目标缓冲区中容纳源缓冲区的内容。实际上，使用 strcpy() 而必须满足的条件经常难以满足，主要原因是它们在本质上不同于 strcpy 的调用。

Recommendations:
永不应该使用那些无法安全使用的函数。如果这些函数中的任何一个出现在新的或是继承代码中，则必须删除该函数并用相应的安全函数进行取代。

不要调用 strcpy() 函数及其类似函数，而是用带有边界的替换函数来取代，如 strncpy()。在 Windows(R) 平台上，考虑使用在 strsafe.h 定义的函数，如按字节形式采用缓冲区大小的 StringCbCopy()，或是按字符形式采用缓冲区大小的 StringCchCopy()。在 BSD Unix 系统上，可以安全地使用 strlcpy()。因为它像 strncpy() 一样，总是以 "\0" 来终止其目标缓冲区。在其他系统上，请使用 strncpy(d, s, SIZE_D) 来替代 strcpy(d, s)，以检查边界是否合适，并防止从 strncpy() 中溢出目标缓冲区。例如，如果 d 是一个对栈分配缓冲区，那么使用 sizeof(d) 就可以计算 SIZE_D 的值。

Tips:
1. 在 Windows 中，将安全性较低的功能（如 strcpy()）替换为安全性较高的功能（如 strcpy_s()）。但应小心谨慎。因为 _s 家族的函数提供的参数验证方法各不相同，所以依靠这些函数可能导致意外的行为。而且，目标缓冲区大小指定不正确仍会导致 buffer overflow 和 null termination 错误。

config.c, line 187 (Dangerous Function: strcpy())

Fortify Priority:	Low	**Folder**	Low
Kingdom:	API Abuse		
Abstract:	无法安全地使用函数 strcpy()。不应该使用此函数。		
Sink:	config.c:187 strcpy()		

```
185                 {
186                     printf("the size of buffer is too small.%d bytes is recommended\n", CONFIG_LINE_BUFFER_SIZE);
187                     goto error;
188                 }
```

config.c, line 217 (Dangerous Function: strcpy())

Fortify Priority:	Low	**Folder**	Low
Kingdom:	API Abuse		
Abstract:	无法安全地使用函数 strcpy()。不应该使用此函数。		
Sink:	config.c:217 strcpy()		

```
215                 //else
216                 //    buf[strlen(buf) - 1] = '\0';
217                 if (buf[strlen(buf) - 1] == '\n' && buf[strlen(buf) - 2] == ']' && strlen(buf) < sizeof(buf))
```

```
218                         buf[strlen(buf) - 2] = '\0';
219                       else if ('[' == buf[strlen(buf) - 1] && strlen(buf) < sizeof(buf))
```

get-abnormal-log.c, line 43 (Dangerous Function: strcpy())

Fortify Priority:	Low	**Folder**	Low
Kingdom:	API Abuse		
Abstract:	无法安全地使用函数 strcpy()。不应该使用此函数。		
Sink:	get-abnormal-log.c:43 strcpy()		

```
41
42                    if (fgets(line, sizeof(line), pout) != NULL) {
43                      strcpy(cmdret, line);
44                      if (cmdret[strlen(cmdret) - 1] == '\n')
45                        cmdret[strlen(cmdret) -1] = '\0';
```

get-abnormal-log.c, line 244 (Dangerous Function: strcpy())

Fortify Priority:	Low	**Folder**	Low
Kingdom:	API Abuse		
Abstract:	无法安全地使用函数 strcpy()。不应该使用此函数。		
Sink:	get-abnormal-log.c:244 strcpy()		

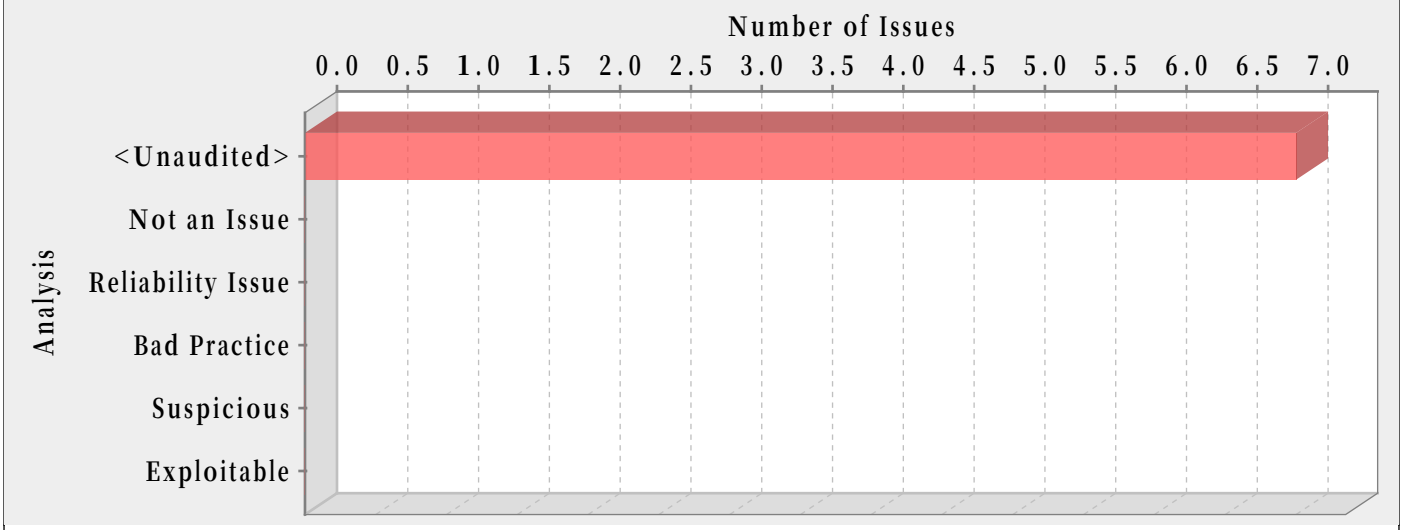
```
242                   }
243
244                   strcpy(DirName, sPathName);
245                   if(DirName[len-1] != '/' && len + 1 < NAME_SIZE)
246                     strcat(DirName, "/");
```

get-abnormal-log.c, line 298 (Dangerous Function: strcpy())

Fortify Priority:	Low	**Folder**	Low
Kingdom:	API Abuse		
Abstract:	无法安全地使用函数 strcpy()。不应该使用此函数。		
Sink:	get-abnormal-log.c:298 strcpy()		

```
296
297                   if (strlen(dst) < sizeof(dir_path))
298                     strcpy(dir_path, dst);
299                   else
300                     return -1;
```

Category: Buffer Overflow (7 Issues)



Abstract:

config.c 中的 config_do_query() 函数可能会在第 163 行中分配的内存边界之外写入数据，这可能会破坏数据、造成程序崩溃或导致恶意代码的执行。

Explanation:

Buffer overflow 可能是人们最熟悉的一种软件安全漏洞。虽然绝大多数软件开发者都知道什么是 Buffer overflow 漏洞，但是无论是对继承下来的或是新开发的应用程序来说，Buffer overflow 攻击仍然是一种最常见的攻击形式。对于这个问题出现的原因，一方面是造成 buffer overflow 漏洞的方式有很多种，另一方面是用于防止 buffer overflow 的技术也容易出错。

在一个典型的 buffer overflow 攻击中，攻击者将数据传送到某个程序，程序会将这些数据储存到一个较小的堆栈缓冲区内。结果，调用堆栈上的信息会被覆盖，其中包括函数的返回指针。数据会被用来设置返回指针的值，这样，当该函数返回时，函数的控制权便会转移给包含在攻击者数据中的恶意代码。

虽然这种类型的堆栈 buffer overflow 在某些平台和开发组织中十分常见，但仍不乏存在其他各种类型的 buffer overflow，其中包括堆 buffer overflow 和 off-by-one 错误等。有关 buffer overflow 如何进行攻击的详细信息，许多优秀的著作都进行了相关介绍，如 Building Secure Software [1]、Writing Secure Code [2] 以及 The Shellcoder's Handbook [3]。

在代码层上，buffer overflow 漏洞通常会违反程序员的各种假设。C 和 C++ 中的很多内存处理函数都没有执行边界检查，因而可以轻易地覆盖缓冲区所操作的、已分配的边界。即使是边界函数（如 strncpy()），使用方式不正确也会引发漏洞。对内存的处理加之有关数据段大小和结构方面所存在种种错误假设，是导致大多数 buffer overflow 漏洞产生的根源。

Buffer overflow 漏洞通常出现在以下代码中：

— 依靠外部的数据来控制行为的代码。

— 受数据属性影响的代码，该数据在代码的临接范围之外执行。

— 代码过于复杂，以致于程序员无法准确预测它的行为。

以下例子分别演示了上面三种情况。

例 1：这是一个关于第二种情况的例子，代码受未在本地校验的数据属性的影响。在本例中，名为 lccopy() 的函数将一个字符串作为其变量，然后返回一个堆分配字符串副本，并将该字符串的所有大写字母转化成了小写字母。因为该函数认为 str 总是比 BUFSIZE 小，所以它不会对输入执行任何边界检查。如果攻击者避开对调用 lccopy() 代码的检查，或者如果更改代码，使得程序员对 str 长度的原有假设与实际不符，那么 lccopy() 就会通过无边界调用 strcpy() 溢出 buf。

```
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); }
```

例 2.a：以下示例代码显示了简单的 buffer overflow，它通常由第一种情况所导致，即依靠外部数据来控制行为的代码。该代码使用 gets() 函数将一个任意大小的数据读取到堆栈缓冲区中。因为没有什么方法可以限制该函数读取数据的量，所以代码的安全性就依赖于用户始终输入比 BUFSIZE 少的字符数量。

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

例 2.b：这一例子表明模仿 C++ 中 gets() 函数的不安全行为是如此的简单，只要通过使用 >> 运算符将输入读取到 char[] 字符串中。

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

例 3：虽然本例中的代码也是依赖于用户输入来控制代码行为，但是它通过使用边界内存复制函数 memcpy() 增加了一个间接级。该函数接受一个目标缓冲区、一个起始缓冲区和要复制的字节数。虽然输入缓冲区由 read() 的边界调用填充，但是 memcpy() 复制的字节数需要由用户指定。

```
...
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);
...
```

注：该类型的 buffer overflow 漏洞（程序可读取数据，然后对剩余数据随后进行的内存操作中的一个数值给予信任）已在图像、音频和其他的文件处理库中频繁地出现。

例 4：以下代码演示了第三种情况，代码过于复杂，以致于程序员无法准确预测它的行为。本代码来自于常用的 libPNG 图象解码器，这种解码器被广泛应用于许多应用程序中，包括 Mozilla 和某些 Internet Explorer 版本。

该代码似乎可以安全地执行边界检查，因为它检测变量长度的大小，该变量长度会在之后用来控制 png_crc_read() 复制的数据量。然而，在测试长度前，该代码会立即对 png_ptr->mode 执行检查，如果检查失败，便会发出一个警告，然后会继续进行处理。因为 length 测试在 else if 模块中进行，如果针对该代码的首次测试失败，那么就不会再测试 length 了，而将其盲目地用于调用 png_crc_read()，因此很容易引起堆栈 buffer overflow。

虽然本例中的代码不是我们所遇见的代码中最复杂的，但是它足以说明为什么要尽可能地降低执行内存操作代码的复杂度。

```
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);
```

例 5：本例同样演示了第三种情况，程序过于复杂，使其暴露出 buffer overflow 的问题。在这种情况下，问题出现的原因在于其中某个函数的接口不明确，而不是代码结构（同上一个例子中描述的情况一样）。

getUserInfo() 函数采用一个定义为多字节字符串的用户名和一个指向用户信息结构的指针，这一结构由该用户的相关信息填充。因为 Windows authentication 中的用户名使用 Unicode，所以 username 参数首先要从多字节字符串转换成 Unicode 字符串。然后，这个函数便会错误地将 unicodeUser 的长度以字节形式而不是字符形式传递出去。调用 MultiByteToWideChar() 可能会把 (UNLEN+1)*sizeof(WCHAR) 宽字符或者 (UNLEN+1)*sizeof(WCHAR)*sizeof(WCHAR) 字节，写到 unicodeUser 数组，该数组仅分配了 (UNLEN+1)*sizeof(WCHAR) 个字节。如果 username 字符串包含了多于 UNLEN 的字符，那么调用 MultiByteToWideChar() 将会溢出 unicodeUser 缓冲区。

```
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:

绝不要使用自身安全性较差的函数，如 gets()，避免使用那些难以安全使用的函数，如 strcpy()。使用相应的边界函数（如 strncpy() 或在 strsafe.h [4] 中定义的 WinAPI 函数）来取代无边界函数（如 strcpy()）。

虽然谨慎使用边界函数能够大大降低 buffer overflow 的风险，但这种移植也不能盲目地进行，而且依靠它自己来确保安全是远远不够的。当您利用内存时（特别是字符串），切记 buffer overflow 漏洞通常会出现在以下代码中：

— 依靠外部的数据来控制行为的代码

— 受数据属性影响的代码，该数据在代码的直接（临界）范围之外执行

— 代码过于复杂，以致于程序员无法准确预测它的行为。

另外，还要考虑到以下原则：

— 永远不要相信外部资源会为内存操作提供正确的控制信息。

— 永远不要相信程序会在执行过程中维护正在处理中的数据所带有的属性。进行操作之前，对这些数据执行健全性检查。

— 限制内存处理和边界检查代码的复杂度。保持这些代码的简单性，并清楚地记录已执行的检查、已测试的假设以及在输入验证失败后，希望程序执行的操作。

— 如果输入的数据太大，截短数据要十分小心，然后再对这些数据继续处理。截短数据会改变输入的含义。

— 不要依赖诸如 StackGuard 之类的工具，或非可执行堆栈来阻止 buffer overflow 漏洞。这些方法不能应对堆 buffer overflow 以及更加敏感的堆栈溢出，该堆栈溢出可以改变控制程序的变量内容。另外，许多这样的方法都很容易破解，即使它们运行比较到位，也只能够找出问题的征兆，而不能发现问题的原因。

Tips:

1. 在 Windows 中，将安全性较低的功能（如 memcpy()）替换为安全性较高的功能（如 memcpy_s()）。但应小心谨慎。因为 _s 家族的函数提供的参数验证方法各不相同，所以依靠这些函数可能导致意外的行为。而且，目标缓冲区大小指定不正确仍会导致 buffer overflow。

config.c, line 187 (Buffer Overflow)

Fortify Priority:	Critical	**Folder**	Critical
Kingdom:	Input Validation and Representation		
Abstract:	config.c 中的 config_do_query() 函数可能会在第 187 行中分配的内存边界之外写入数据，这可能会破坏数据、造成程序崩溃或导致恶意代码的执行。		

Source: config.c:131 fgets()

```
129    {
130        printf("fail to fseek\n");
131        //printf("fail to fseek:%s\n", strerror(errno));
132        goto error;
133    }
```

Sink: config.c:187 strcpy()

```
185                {
186                    printf("the size of buffer is too small.%d bytes is recommended\n", CONFIG_LINE_BUFFER_SIZE);
187                    goto error;
188                }
```

config.c, line 217 (Buffer Overflow)

Fortify Priority:	Critical	**Folder**	Critical
Kingdom:	Input Validation and Representation		
Abstract:	config.c 中的 config_do_query() 函数可能会在第 217 行中分配的内存边界之外写入数据，这可能会破坏数据、造成程序崩溃或导致恶意代码的执行。		

Source: config.c:131 fgets()

```
129    {
130        printf("fail to fseek\n");
131        //printf("fail to fseek:%s\n", strerror(errno));
132        goto error;
133    }
```

Sink: config.c:217 strcpy()

```
215                //else
216                //    buf[strlen(buf) - 1] = '\0';
217                if (buf[strlen(buf) - 1] == '\n' && buf[strlen(buf) - 2] == ']' && strlen(buf) < sizeof(buf))
218                    buf[strlen(buf) - 2] = '\0';
219                else if ('[' == buf[strlen(buf) - 1] && strlen(buf) < sizeof(buf))
```

config.c, line 163 (Buffer Overflow)

Fortify Priority:	High	**Folder**	High
Kingdom:	Input Validation and Representation		

Abstract:	config.c 中的 config_do_query() 函数可能会在第 163 行中分配的内存边界之外写入数据，这可能会破坏数据、造成程序崩溃或导致恶意代码的执行。
Source:	config.c:131 fgets()

```
129   {
130     printf("fail to fseek\n");
131     //printf("fail to fseek:%s\n", strerror(errno));
132     goto error;
133   }
```

Sink:	config.c:163 Assignment to val[]()

```
161             }
162
163             //if (strstr(val, "\r"))
164             //    val[strlen(val) - 2] = '\0';
165             //else if (strstr(val, "\n"))
```

config.c, line 211 (Buffer Overflow)

Fortify Priority:	High	**Folder**	High
Kingdom:	Input Validation and Representation		
Abstract:	config.c 中的 config_do_query() 函数可能会在第 211 行中分配的内存边界之外写入数据，这可能会破坏数据、造成程序崩溃或导致恶意代码的执行。		
Source:	config.c:131 fgets()		

```
129   {
130     printf("fail to fseek\n");
131     //printf("fail to fseek:%s\n", strerror(errno));
132     goto error;
133   }
```

Sink:	config.c:211 Assignment to buf[]()

```
209         }
210
211         //if (strstr(buf, "\r"))
212         //    buf[strlen(buf) - 3 ] = '\0';
213         //else if (strstr(buf, "\n"))
```

config.c, line 213 (Buffer Overflow)

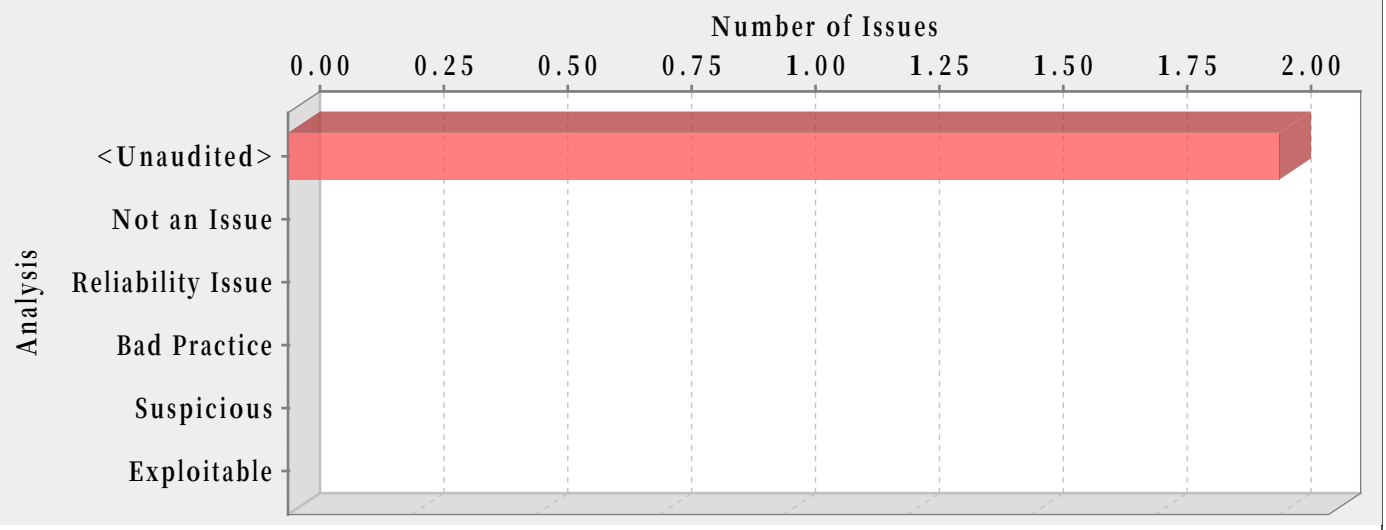
Fortify Priority:	High	**Folder**	High
Kingdom:	Input Validation and Representation		
Abstract:	config.c 中的 config_do_query() 函数可能会在第 213 行中分配的内存边界之外写入数据，这可能会破坏数据、造成程序崩溃或导致恶意代码的执行。		
Source:	config.c:131 fgets()		

```
129   {
130     printf("fail to fseek\n");
131     //printf("fail to fseek:%s\n", strerror(errno));
132     goto error;
133   }
```

Sink:	config.c:213 Assignment to buf[]()

```
211         //if (strstr(buf, "\r"))
212         //    buf[strlen(buf) - 3 ] = '\0';
213         //else if (strstr(buf, "\n"))
214         //    buf[strlen(buf) - 2 ] = '\0';
215         //else
```

Category: Path Manipulation (2 Issues)



Abstract:
攻击者可以控制 get-abnormal-log.c 中第 216 行的 mkdir() file system 路径参数，借此访问或修改其他受保护的文件。

Explanation:
当满足以下两个条件时，就会产生 path manipulation 错误：

1. 攻击者能够指定某一 file system 操作中所使用的路径。

2. 攻击者可以通过指定特定资源来获取某种权限，而这种权限在一般情况下是不可能获得的。

例如，在某一程序中，攻击者可以获得特定的权限，以重写指定的文件或是在其控制的配置环境下运行程序。

例 1：以下代码使用来自 CGI 请求的输入来创建一个文件名。程序员没有考虑到攻击者可能使用像 "../../apache/conf/httpd.conf" 一样的文件名，从而导致应用程序删除特定的配置文件。

```
char* rName = getenv("reportName");
...
unlink(rName);
```

例 2：以下代码使用来自于命令行的输入来决定该打开哪个文件，并返回到用户。如果程序在一定的权限下运行，并且恶意用户能够创建到文件的软链接，那么他们可以使用程序来读取系统中任何文件的开始部分。

```
ifstream ifs(argv[0]);
string s;
ifs >> s;
cout << s;
```

Recommendations:
防止 path manipulation 的最佳方法是采用一些间接手段：例如创建一份合法资源名的列表，并且规定用户只能选择其中的文件名。通过这种方法，用户就不能直接由自己来指定资源的名称了。

但在某些情况下，这种方法并不可行，因为这样一份合法资源名的列表过于庞大、难以跟踪。因此，程序员通常在这种情况下采用黑名单的办法。在输入之前，黑名单会有选择地拒绝或避免潜在的危险字符。但是，任何这样一份黑名单都不可能是完整的，而且将随着时间的推移而过时。更好的方法是创建一份白名单，允许其中的字符出现在资源名称中，且只接受完全由这些被认可的字符组成的输入。

Tips:
1. 程序在执行输入验证时，应确保该验证正确无误，且使用 HP Fortify Custom Rules Editor（HP Fortify 自定义规则编辑器）为该验证例程创建适当的规则。

2. 执行本身有效的黑名单是一件非常困难的事情，因此，如果验证逻辑完全依赖于黑名单方法，那么有必要对这种逻辑进行质疑。鉴于不同类型的输入编码以及各种元字符集在不同的操作系统、数据库或其他资源中可能有不同的含义，确定随着需求的不断变化，黑名单能否方便、正确、完整地进行更新。

get-abnormal-log.c, line 216 (Path Manipulation)

Fortify Priority:	Critical	**Folder**	Critical
Kingdom:	Input Validation and Representation		
Abstract:	攻击者可以控制 get-abnormal-log.c 中第 216 行的 mkdir() file system 路径参数，借此访问或修改其他受保护的文件。		

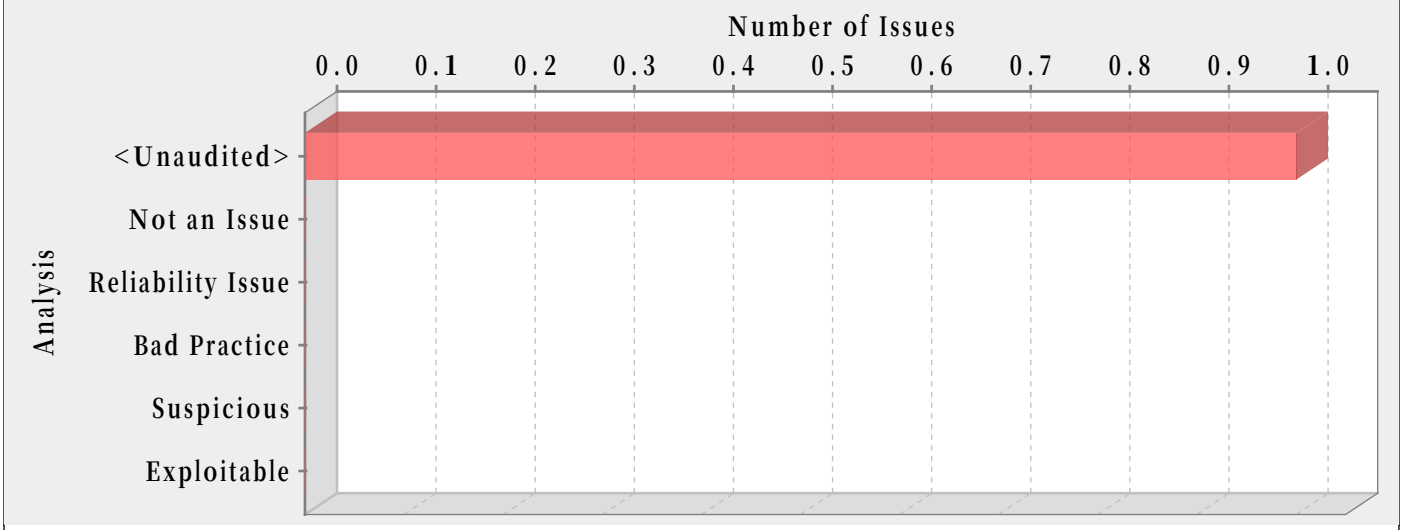
Source:	config.c:131 fgets()
129	{
130	printf("fail to fseek\n");
131	//printf("fail to fseek:%s\n", strerror(errno));
132	goto error;
133	}
Sink:	get-abnormal-log.c:216 mkdir()
214	{
215	int ret = 0;
216	ret = mkdir(dirname, 0755);
217	if (ret == -1) {
218	perror("mkdir");

get-abnormal-log.c, line 216 (Path Manipulation)

Fortify Priority:	Critical	**Folder**	Critical
Kingdom:	Input Validation and Representation		
Abstract:	攻击者可以控制 get-abnormal-log.c 中第 216 行的 mkdir() file system 路径参数，借此访问或修改其他受保护的文件。		

Source:	get-abnormal-log.c:1082 main(1)
1080	}
1081	
1082	int main(int argc, const char *argv[])
1083	{
1084	int ret = 0;
Sink:	get-abnormal-log.c:216 mkdir()
214	{
215	int ret = 0;
216	ret = mkdir(dirname, 0755);
217	if (ret == -1) {
218	perror("mkdir");

Category: Command Injection (1 Issues)



Abstract:

get-abnormal-log.c 中的 getlocaltime() 函数会调用第 35 行的 popen() 来执行命令。这可能会允许攻击者注入恶意的命令。

Explanation:

Command Injection 漏洞主要表现为以下两种形式：

－ 攻击者能够篡改程序执行的命令：攻击者直接控制了所执行的命令。

－ 攻击者能够篡改命令的执行环境：攻击者间接地控制了所执行的命令。

在这种情况下，我们主要关注第二种情形，即攻击者能够通过篡改一个环境变量或预先在搜索路径中进行恶意输入，来篡改命令的含义。这种形式的 Command Injection 漏洞在以下情况下发生：

1. 攻击者篡改某一应用程序的环境。

2. 应用程序在没有指明绝对路径，或者没有检验所执行的二进制代码的情况下就执行命令。

3. 通过命令的执行，应用程序会授予攻击者一种原本不该拥有的特权或能力。

例 1：这个例子演示了当攻击者能够篡改命令的解析方式时，会发生什么情况。这段代码来自于一个基于 Web 的 CGI 功能模块，用户可以利用这个模块修改他们的密码。通过网络信息服务执行的密码更新过程包括在 /var/yp 目录中运行 make。请注意，由于程序更新了密码记录，因此它已按照 setuid root 安装。

程序会调用 make，如下所示：

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

因为这个例子中的命令是采用硬编码编写的，所以攻击者无法控制传送到 system() 的参数。但是，因为程序没有指定 make 的绝对路径，而且没有在调用命令之前清除任何环境变量，所以攻击者能够篡改 $PATH 变量，从而指向一个名为 make 的恶意的二进制代码，并通过 shell 提示符来执行 CGI 脚本。而且，因为程序已按照 setuid root 安装，所以攻击者的 make 目前会在 root 的权限下执行。

在 Windows 中，还存在其他风险。

例 2：直接或通过调用 _spawn() 家族中的某项函数调用 CreateProcess() 时，如果可执行文件或路径中存在空格，必须谨慎操作。

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

CreateProcess() 解析空格时，操作系统尝试执行的第一个可执行文件将是 Program.exe，而不是 MyApplication.exe。因此，如果攻击者能够在系统上安装名称为 Program.exe 的恶意应用程序，任何使用 Program Files 目录错误调用 CreateProcess() 的程序将运行此恶意应用程序，而非原本期望的应用程序。

环境在程序的系统命令执行中扮演了一个十分重要的角色。由于诸如 system() 和 exec() 之类的函数利用程序调用这些函数的环境，因此攻击者有可能创造干扰这些调用行为。

Recommendations:

留意外部环境，看环境会对您所执行命令的行为产生何种影响。特别要注意 $PATH、$LD_LIBRARY_PATH 和 $IFS 变量在 Unix 和 Linux 机器中的使用方式。

请注意，Windows APIs 使用的是一个特殊的搜索顺序，它不仅仅基于一系列目录，而且还基于在没有特别指定时自动添加的文件扩展名列表。例如，对于 _spawn() 家族中的函数，如果命令名称参数没有文件扩展名或不以句号结束，那么它将尝试通过以下的顺序执行文件扩展名：首先 .com，其次 .exe，然后 .bat，最后 .cmd。此外，由于执行命令时，函数将解析代表可执行文件和路径的参数中的空格，这使得 Windows 中存在其他风险。

例 3：以下代码将例 2 重写，通过对可执行路径使用引号以避免无意中执行恶意应用程序。

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

另一种方法也能达到同样的效果，即将可执行文件的名称作为第一个参数传递，而不是传递 NULL。

尽管可能无法完全阻止强大的攻击者为了控制程序执行的命令而对系统进行的攻击，但只要程序执行外部命令，就务必使用最小授权原则：不给予超过执行该命令所必需的权限。

Tips:

1. 在 Windows 中，如果执行的命令是 Windows 的内部命令，则报告的问题并不会受到关注。内部命令并不驻留在磁盘上。相反，它们驻留在 COMMAND.COM 中，当计算机系统启动时，它们会加载到内存中。一系列内部命令包括：BREAK、CALL、CHCP、CHDIR(CD)、CLS、COPY、CTTY、DATE、DEL(ERASE)、DIR、ECHO、EXIT、FOR、GOTO、IF、MKDIR(MD)、PATH、PAUSE、PROMPT、REM、RENAME(REN)、RMDIR(RD)、SET、SHIFT、TIME、TYPE、VER、VERIFY、VOL。有关特定于您系统的内部命令的最新列表，请查阅您的系统文档。

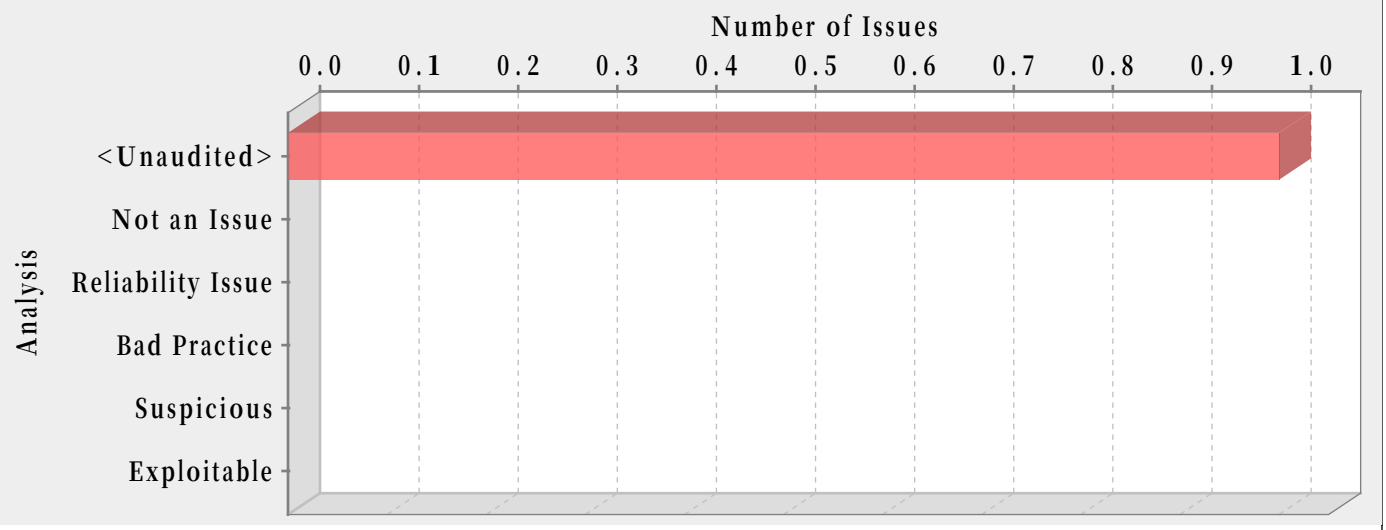
get-abnormal-log.c, line 35 (Command Injection)

Fortify Priority:	Low	**Folder**	Low
Kingdom:	Input Validation and Representation		
Abstract:	get-abnormal-log.c 中的 getlocaltime() 函数会调用第 35 行的 popen() 来执行命令。这可能会允许攻击者注入恶意的命令。		

Sink: get-abnormal-log.c:35 popen(0)

```
33    char line[256] = {0};
34
35    pout = popen(cmd, "r");
36    if (!pout) {
37       pr_err("popen error\n");
```

Category: System Information Leak: Internal (1 Issues)



Abstract:

config.c 中的 config_do_query() 函数通过调用第 126 行的 printf() 来揭示系统数据或调试信息。由 printf() 揭示的信息有助于攻击者制定攻击计划。

Explanation:

通过日志或打印功能将系统数据或调试信息发送到本地文件、控制台或屏幕时，就会发生内部信息泄露。

示例：以下代码将路径变量输出到标准的错误流：

```
char* path = getenv("PATH");
...
sprintf(stderr, "cannot find exe on path %s\n", path);
```

依据这一系统配置，该信息可转储到控制台，写成日志文件，或者显示给远程用户。在某些情况下，这个错误消息正好可以准确地告诉攻击者系统被入侵的可能性有多大。例如，一个数据库错误消息可以揭示应用程序容易受到 SQL Injection 攻击。其他的错误消息可以揭示有关该系统的更多间接线索。在上述例子中，搜索路径可能会暗示操作系统的类型、系统上安装了哪些应用程序，以及管理员在配置应用程序时做了哪些方面的努力。

Recommendations:

编写错误消息时，始终要牢记安全性。在编码的过程中，尽量避免使用繁复的消息，提倡使用简短的错误消息。限制生成与存储繁复的输出数据将有助于管理员和程序员诊断问题的所在。此外，还要留意有关调试的跟踪信息，有时它可能出现在不明显的位置（例如嵌入在错误页 HTML 代码的注释行中）。

即便是并未揭示栈踪迹或数据库转储的简短错误消息，也有可能帮助攻击者发起攻击。例如，"Access Denied"（拒绝访问）消息可以揭示系统中存在一个文件或用户。

Tips:

1. 不要依赖于封装器脚本、组织内部的 IT 策略或是思维敏捷的系统管理员来避免 System Information Leak 漏洞。编写安全的软件才是关键。

2. 这类漏洞并不适用于所有类型的程序。例如，如果您在一个客户机上执行应用程序，而攻击者已经获取了该客户机上的系统信息，或者如果您仅把系统信息打印到一个可信赖的日志文件中，就可以使用 AuditGuide 来过滤这一类别。

config.c, line 126 (System Information Leak: Internal)

Fortify Priority:	Low	**Folder**	Low
Kingdom:	Encapsulation		
Abstract:	config.c 中的 config_do_query() 函数通过调用第 126 行的 printf() 来揭示系统数据或调试信息。由 printf() 揭示的信息有助于攻击者制定攻击计划。		

Source: config.c:126 strerror()

```
124     }
125
126     memset(buffer, 0, size);
127
128     if (fseek(fd, 0, SEEK_SET) == -1)
```

Sink: config.c:126 printf()

```
124     }
125
126     memset(buffer, 0, size);
```

```
127
128                     if (fseek(fd, 0, SEEK_SET) == -1)
```