Hewlett Packard Enterprise

HPE Security Fortify Standalone Report Generator

Developer Workbook

testcase

FORTIFY

Table of Contents

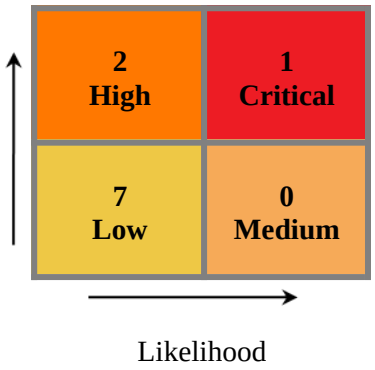
Executive Summary

This workbook is intended to provide all necessary details and information for a developer to understand and remediate the different issues discovered during the testcase project audit. The information contained in this workbook is targeted at project managers and developers.

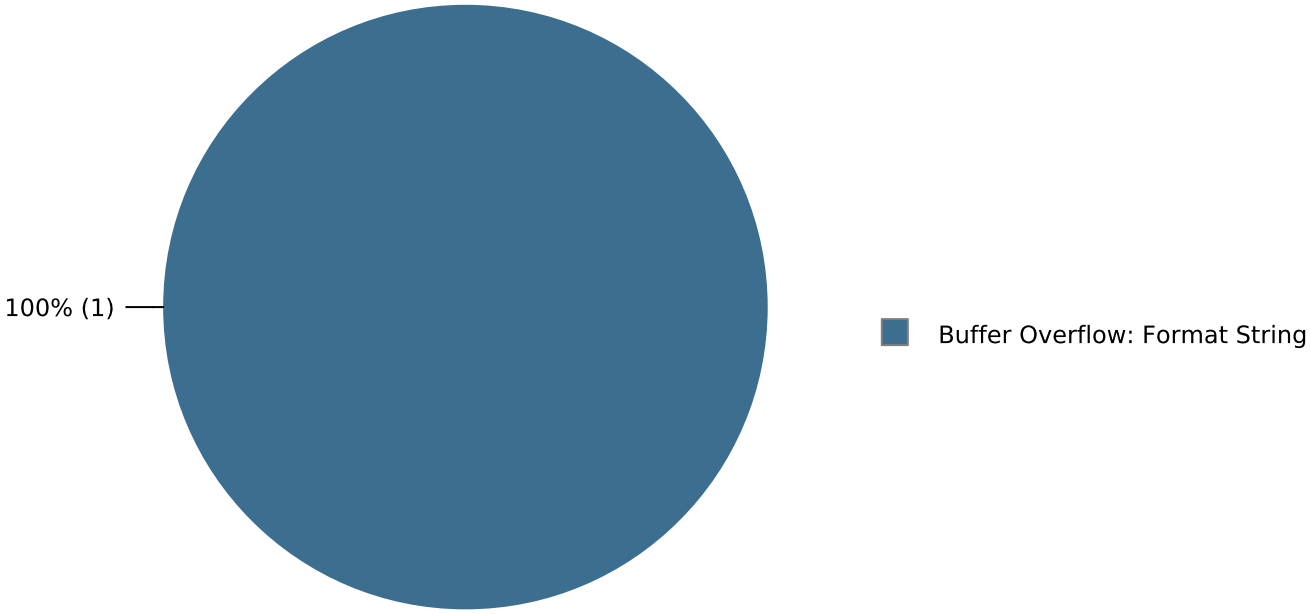
This section provides an overview of the issues uncovered during analysis.

Project Name:	testcase
Project Version:	
SCA:	Results Present
WebInspect:	Results Not Present
SecurityScope:	Results Not Present
Other:	Results Not Present

Issues by Priority



Top Ten Critical Categories



Project Description

This section provides an overview of the HPE Security Fortify scan engines used for this project, as well as the project meta-information.

SCA

Date of Last Analysis:	Oct 8, 2018, 5:14 PM	**Engine Version:**	17.10.0156
Host Name:	ubuntu	**Certification:**	VALID
Number of Files:	37	**Lines of Code:**	666

Issue Breakdown by Fortify Categories

The following table depicts a summary of all issues grouped vertically by Fortify Category. For each category, the total number of issues is shown by Fortify Priority Order, including information about the number of audited issues.

Category	Fortify Priority (audited/total)				Total Issues
	Critical	High	Medium	Low	
Buffer Overflow: Format String	0 / 1	0	0	0	0 / 1
Format String: Argument Type Mismatch	0	0 / 1	0	0	0 / 1
Memory Leak	0	0 / 1	0	0	0 / 1
Poor Style: Redundant Initialization	0	0	0	0 / 2	0 / 2
System Information Leak	0	0	0	0 / 1	0 / 1
Uninitialized Variable	0	0	0	0 / 4	0 / 4

Results Outline

Buffer Overflow: Format String (1 issue)

Abstract

The program uses an improperly bounded format string, allowing it to write outside the bounds of allocated memory. This behavior could corrupt data, crash the program, or lead to 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 exceed 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.

In this case, an improperly constructed format string causes the program to write beyond the bounds of allocated memory.

Example: The following code overflows `c` because the `double` type requires more space than is allocated for `c`.

```
void formatString(double d) {
    char c;

    scanf("%d", &c)
}
```

Recommendation

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.

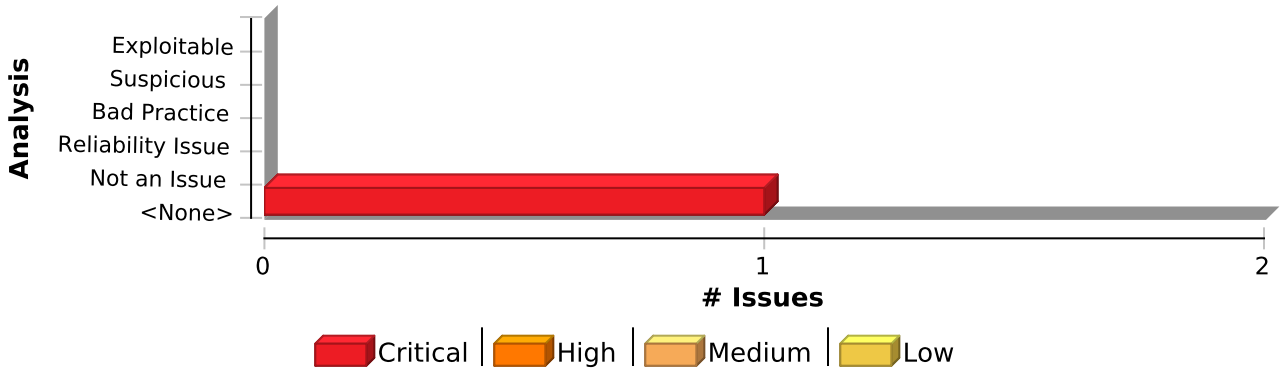
- Never trust that properties about the data your program is manipulating will be maintained throughout the program. Sanity check data before you operate on it.

- 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.

Issue Summary



Engine Breakdown

	SCA	WebInspect	SecurityScope	Total
Buffer Overflow: Format String	1	0	0	1
Total	**1**	**0**	**0**	**1**

Buffer Overflow: Format String — Critical

Package: <none>

sys_call_2.c, line 11 (Buffer Overflow: Format String) — Critical

Issue Details

Kingdom: Input Validation and Representation
Scan Engine: SCA (Data Flow)

Source Details

Buffer Overflow: Format String — Critical

Package: <none>

sys_call_2.c, line 11 (Buffer Overflow: Format String) — Critical

Source: Buffer h Declared
From: main
File: sys_call_2.c:7

```
4 int main(void)
5 {
6  char ptr_h;
7  char h[64];
8
9  ptr_h = getenv("HOME"); // value of HOME could be > 64, need size guard
10  if (ptr_h != NULL) {
```

Sink Details

Sink: sprintf()
Enclosing Method: main()
File: sys_call_2.c:11
Taint Flags: ENVIRONMENT, NULL_TERMINATED

```
8
9  ptr_h = getenv("HOME"); // value of HOME could be > 64, need size guard
10  if (ptr_h != NULL) {
11  sprintf(h, "your home dir is: %s :", ptr_h);
12  printf("%s\n",h);
13  }
14  return 0;
```

Format String: Argument Type Mismatch (1 issue)

Abstract

The program uses an improperly constructed format string that contains conversion specifiers that do not align with the types of the arguments passed to the function. Incorrect format strings can lead the program to convert values incorrectly and potentially read or write outside the bounds of allocated memory, which can introduce incorrect behavior or crash the program.

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 exceed 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.

In this case, an improperly constructed format string causes the program to improperly convert data values or to access values outside the bounds of allocated memory.

Example: The following code incorrectly converts `f` from a float using a `%d` format specifier.

```
void ArgTypeMismatch(float f, int d, char *s, wchar *ws) {
   char buf[1024];
   sprintf(buf, "Wrong type of %d", f);
   ...
}
```

Recommendation

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.

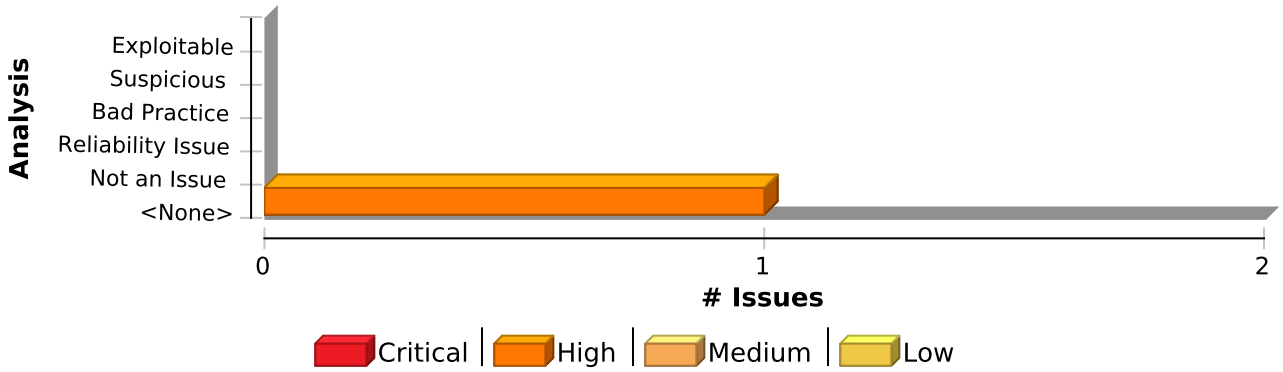
- Never trust that properties about the data your program is manipulating will be maintained throughout the program. Sanity check data before you operate on it.

- 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.

Issue Summary



Engine Breakdown

	SCA	WebInspect	SecurityScope	Total
Format String: Argument Type Mismatch	1	0	0	1
Total	**1**	**0**	**0**	**1**

Format String: Argument Type Mismatch — High

Package: <none>

sys_call_2.c, line 11 (Format String: Argument Type Mismatch) — High

Issue Details

Kingdom: Code Quality
Scan Engine: SCA (Semantic)

Sink Details

Format String: Argument Type Mismatch	High
Package: <none>	
sys_call_2.c, line 11 (Format String: Argument Type Mismatch)	**High**

Sink: sprintf()
Enclosing Method: main()
File: sys_call_2.c:11
Taint Flags:

```
8
9  ptr_h = getenv("HOME"); // value of HOME could be > 64, need size guard
10 if (ptr_h != NULL) {
11 sprintf(h, "your home dir is: %s :", ptr_h);
12 printf("%s\n",h);
13 }
14 return 0;
```

Memory Leak (1 issue)

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 might 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;
}
```

Recommendation

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.

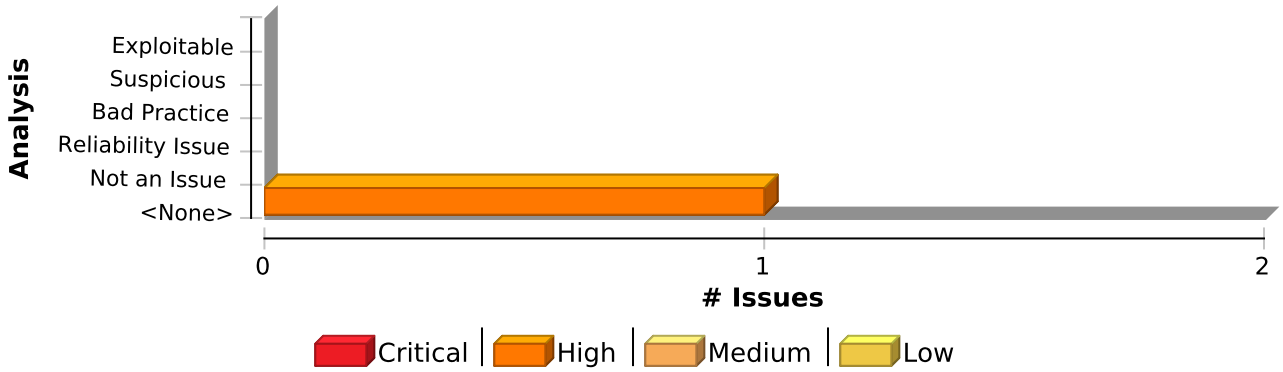
One good pattern for addressing the error handling mistake in the example is to use forward-reaching `goto` statements so that the function has a single well-defined region for handling errors, as follows:

```
  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;
}
```

Issue Summary



Engine Breakdown

	SCA	WebInspect	SecurityScope	Total
Memory Leak	1	0	0	1
Total	**1**	**0**	**0**	**1**

Memory Leak — High

Package: <none>

sys_call_1.c, line 27 (Memory Leak) — High

Issue Details

Kingdom: Code Quality
Scan Engine: SCA (Control Flow)

Sink Details

Sink: lcl_s = malloc(...)
Enclosing Method: make_s()
File: sys_call_1.c:27
Taint Flags:

```
24 struct s *make_s(char *d)
25 {
26  struct s *lcl_s;
27  lcl_s = (struct s *)malloc(sizeof(lcl_s));
28
29  if (lcl_s != NULL) {
30  return NULL;
```

Poor Style: Redundant Initialization (2 issues)

Abstract

The variable's value is assigned but never used, making it a dead store.

Explanation

This variable's initial value is not used. After initialization, the variable is either assigned another value or goes out of scope.

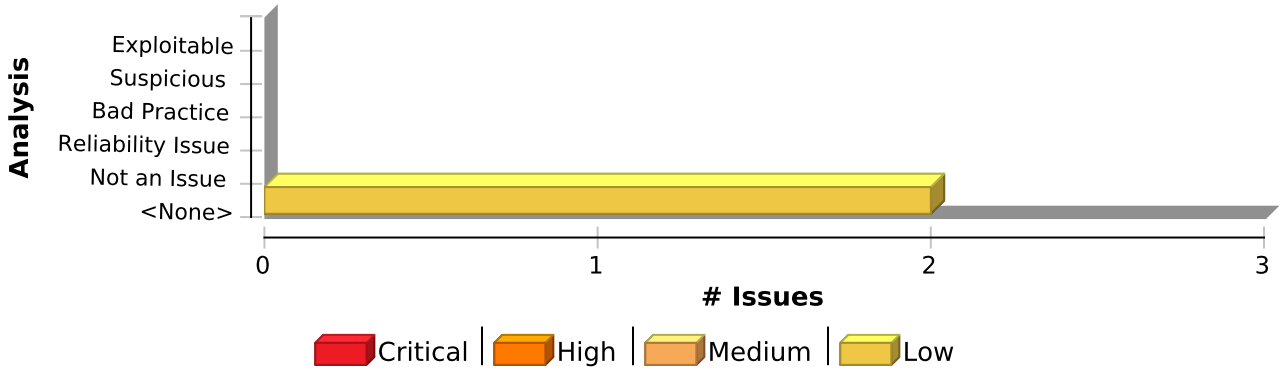
Example: The following code excerpt assigns to the variable `r` and then overwrites the value without using it.

```
int r = getNum();
r = getNewNum(buf);
```

Recommendation

Remove unnecessary assignments in order to make the code easier to understand and maintain.

Issue Summary



Engine Breakdown

	SCA	WebInspect	SecurityScope	Total
Poor Style: Redundant Initialization	2	0	0	2
Total	**2**	**0**	**0**	**2**

Poor Style: Redundant Initialization — Low

Package: <none>

asm.c, line 3 (Poor Style: Redundant Initialization) — Low

Issue Details

Kingdom: Code Quality
Scan Engine: SCA (Structural)

Sink Details

Poor Style: Redundant Initialization — Low

Package: <none>

asm.c, line 3 (Poor Style: Redundant Initialization) — Low

Sink: VariableAccess: c
Enclosing Method: asm_add()
File: asm.c:3
Taint Flags:

```
1 int asm_add(int a) {
2  int b;
3  int c = a;
4  asm ("add %1, %0\n" : "+r"(c) : "r"(b));
5  return c;
6 }
7
```

sys_call_1.c, line 8 (Poor Style: Redundant Initialization) — Low

Issue Details

Kingdom: Code Quality
Scan Engine: SCA (Structural)

Sink Details

Sink: VariableAccess: c
Enclosing Method: foo()
File: sys_call_1.c:8
Taint Flags:

```
5 {
6  FILE *fp;
7
8  int c = getc(fp = fopen(f, "r"));
9
10  if (feof(stdin) || ferror(stdin)) {
11  exit(1);
```

System Information Leak (1 issue)

Abstract

Revealing system data or debugging information helps an adversary learn about the system and form a plan of attack.

Explanation

An information leak occurs when system data or debugging information leaves the program through an output stream or logging function.

Example: The following code prints the path environment variable to the standard error stream:

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

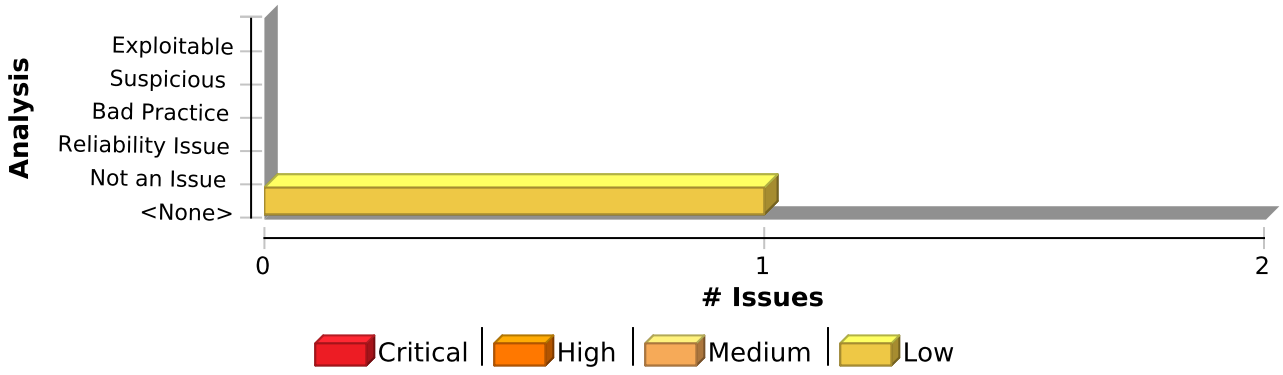
Depending upon the system configuration, this information can be dumped to a console, written to a log file, or exposed to a remote user. In some cases the error message tells the attacker precisely what sort of an attack the system will be vulnerable to. For example, a database error message can reveal that the application is vulnerable to a SQL injection attack. Other error messages can reveal more oblique clues about the system. In the example above, the search path could imply information about the type of operating system, the applications installed on the system, and the amount of care that the administrators have put into configuring the program.

Recommendation

Write error messages with security in mind. In production environments, turn off detailed error information in favor of brief messages. Restrict the generation and storage of detailed output that can help administrators and programmers diagnose problems. Be careful, debugging traces can sometimes appear in non-obvious places (embedded in comments in the HTML for an error page, for example).

Even brief error messages that do not reveal stack traces or database dumps can potentially aid an attacker. For example, an "Access Denied" message can reveal that a file or user exists on the system.

Issue Summary



Engine Breakdown

	SCA	WebInspect	SecurityScope	Total
System Information Leak	1	0	0	1
Total	**1**	**0**	**0**	**1**

System Information Leak — Low

Package: <none>

sys_call_2.c, line 12 (System Information Leak) — Low

Issue Details

Kingdom: Encapsulation
Scan Engine: SCA (Data Flow)

Source Details

Source: getenv()
From: main
File: sys_call_2.c:9

```
6  char ptr_h;
7  char h[64];
8
9  ptr_h = getenv("HOME"); // value of HOME could be > 64, need size guard
10 if (ptr_h != NULL) {
11 sprintf(h, "your home dir is: %s :", ptr_h);
12 printf("%s\n",h);
```

Sink Details

Sink: printf()
Enclosing Method: main()
File: sys_call_2.c:12
Taint Flags: ENVIRONMENT, NULL_TERMINATED

```
9  ptr_h = getenv("HOME"); // value of HOME could be > 64, need size guard
10 if (ptr_h != NULL) {
11 sprintf(h, "your home dir is: %s :", ptr_h);
12 printf("%s\n",h);
13 }
14 return 0;
15 }
```

Uninitialized Variable (4 issues)

Abstract

The program can potentially use a variable before it has been initialized.

Explanation

Stack variables in C and C++ are not initialized by default. Their initial values are determined by whatever happens to be in their location on the stack at the time the function is invoked. Programs should never use the value of an uninitialized variable.

It is not uncommon for programmers to use an uninitialized variable in code that handles errors or other rare and exceptional circumstances. Uninitialized variable warnings can sometimes indicate the presence of a typographic error in the code.

Example 1: The following switch statement is intended to set the values of the variables `aN` and `bN`, but in the default case, the programmer has accidentally set the value of `aN` twice.

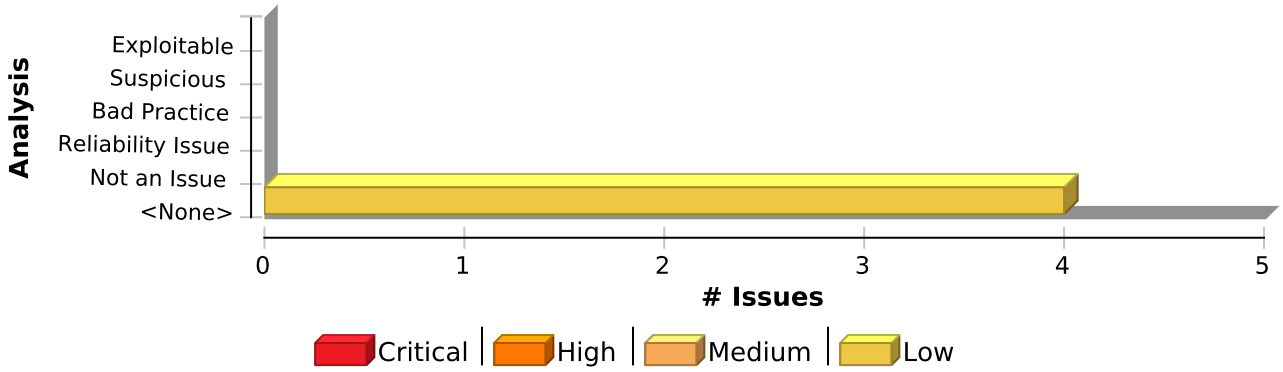
```
switch (ctl) {
  case -1:
    aN = 0; bN = 0;
    break;
  case 0:
    aN = i; bN = -i;
    break;
  case 1:
    aN = i + NEXT_SZ; bN = i - NEXT_SZ;
    break;
  default:
    <b>aN = -1; aN = -1;</b>
    break;
}
```

Most uninitialized variable issues result in general software reliability problems, but if attackers can intentionally trigger the use of an uninitialized variable, they might be able to launch a denial of service attack by crashing the program. Under the right circumstances, an attacker may be able to control the value of an uninitialized variable by affecting the values on the stack prior to the invocation of the function.

Recommendation

Before a variable is used, initialize it.

Issue Summary



Engine Breakdown

	SCA	WebInspect	SecurityScope	Total
Uninitialized Variable	4	0	0	4
Total	**4**	**0**	**0**	**4**

Uninitialized Variable — Low

Package: <none>

clang_false_positive.c, line 18 (Uninitialized Variable) — Low

Issue Details

Kingdom: Code Quality
Scan Engine: SCA (Control Flow)

Sink Details

Sink: q : Variable q used without being initialized
Enclosing Method: memmodel_clang1()
File: clang_false_positive.c:18
Taint Flags:

```
15 p = &buf;
16 // if (buf.data[1] == 1) // if use this form, Clang is fine
17 if (p->data[1] == 1)
18 p->data[0] = *q; // not null pointer deref (false positive for Clang)
19
20 return p->data[0];
21 }
```

clang_non_npd.c, line 18 (Uninitialized Variable) — Low

Issue Details

Kingdom: Code Quality
Scan Engine: SCA (Control Flow)

Sink Details

Sink: q : Variable q used without being initialized
Enclosing Method: memmodel_clang2()
File: clang_non_npd.c:18
Taint Flags:

Uninitialized Variable — Low

Package: <none>

clang_non_npd.c, line 18 (Uninitialized Variable) — Low

```
15  p = &buf;
16  // if (buf.data[1] == 1) // if use this form, Clang is fine
17  if (buf.data[1] == 1)
18  p->data[0] = *q; // not null pointer deref (false positive for Clang)
19
20  return p->data[0];
21  }
```

asm.c, line 4 (Uninitialized Variable) — Low

Issue Details

Kingdom: Code Quality
Scan Engine: SCA (Control Flow)

Sink Details

Sink: b : Variable b used without being initialized
Enclosing Method: asm_add()
File: asm.c:4
Taint Flags:

```
1 int asm_add(int a) {
2  int b;
3  int c = a;
4  asm ("add %1, %0\n" : "+r"(c) : "r"(b));
5  return c;
6 }
7
```

app_uart_fifo_multi_path.c, line 37 (Uninitialized Variable) — Low

Issue Details

Kingdom: Code Quality
Scan Engine: SCA (Control Flow)

Sink Details

Sink: err_code : Variable err_code used without being initialized
Enclosing Method: app_uart_putall()
File: app_uart_fifo_multi_path.c:37
Taint Flags:

```
34 uint32_t app_uart_putall(void)
35 {
36  uint32_t err_code;
37  if (err_code == ((0x0) + 0))
38  {
39  if (!nrf_drv_uart_tx_in_progress(&app_uart_inst))
```

Uninitialized Variable	Low
Package: <none>	
app_uart_fifo_multi_path.c, line 37 (Uninitialized Variable)	**Low**

```
40 {
```