FORTIFY®

28/04/2013

Executive Summary

Issues Overview

On 28/04/2013, a source code review was performed over the tor code base. 315 files, 47.363 LOC (Executable) were scanned and reviewed for defects that could lead to potential security vulnerabilities. A total of 1863 reviewed findings were uncovered during the analysis.

Issues by Fortify Priority Order

High	1630
Low	206
Critical	27

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: /root/tor-0.2.3.25/src

Number of Files: 315

Lines of Code: 47363

Build Label: <No Build Label>

Scan Information

Scan time: 02:39:41

SCA Engine version: 5.10.2.0044

Machine Name: debian

Username running scan: root

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:

Private Information:

null.null.null

System Information:

null.null.geteuid

null.null.getgid

null.null.gethostname

null.null.getrlimit

null.null.getuid

null.null.strerror

null.null.uname

Filter Set Summary

Current Enabled Filter Set:

Security Auditor 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

Audit Guide Summary

Audit guide not enabled

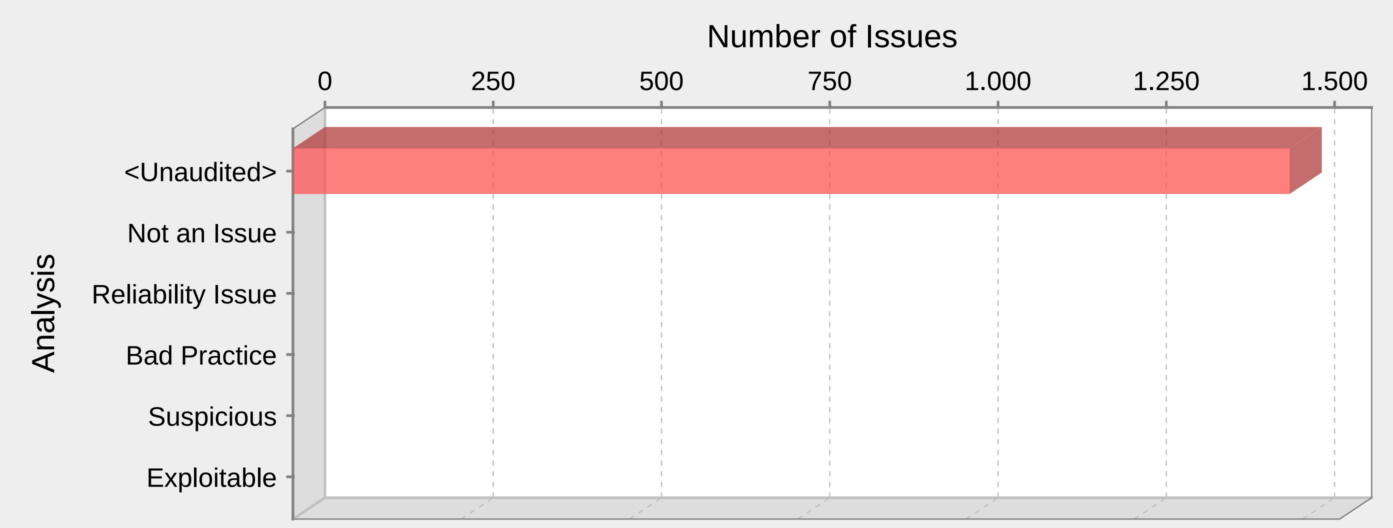
Results Outline

Overall number of results

The scan found 1863 issues.

Vulnerability Examples by Category

Category: Memory Leak (1481 Issues)



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;
}
```

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:

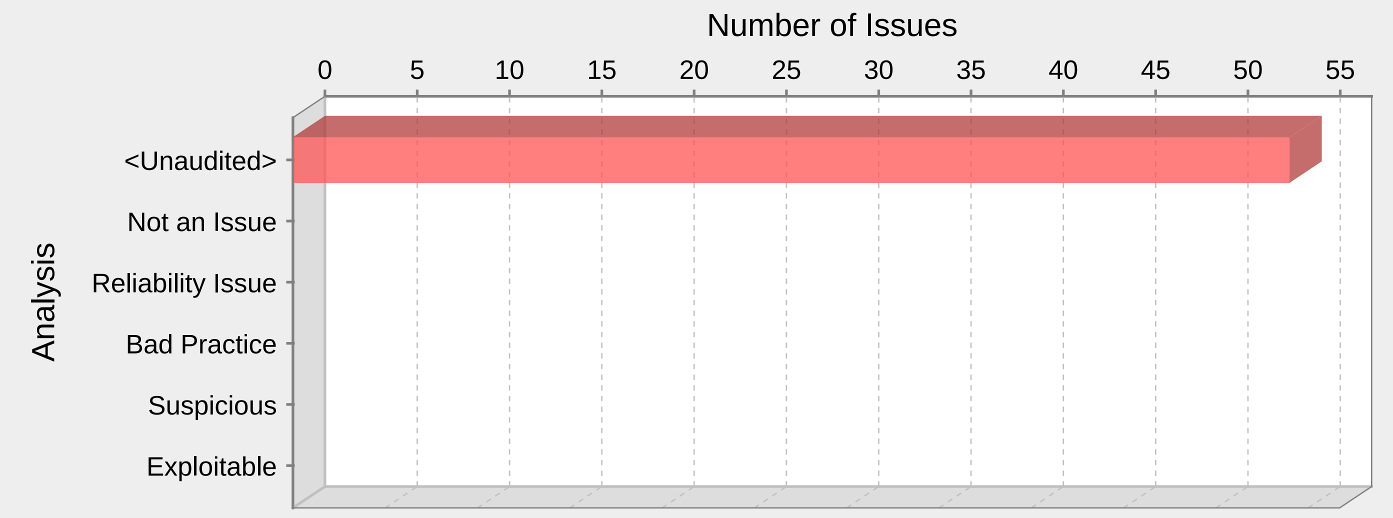
1. Managed pointer objects, such as C++ auto_ptr and Boost smart pointers, are used to ensure that referenced memory allocations are freed. However, memory leaks can still occur when auto_ptrs or certain types of Boost smart pointers are used to reference arrays, Boost array pointers reference individual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

address.c, line 585 (Memory Leak)

Fortify Priority:	High	**Folder**	High
Kingdom:	Code Quality		
Abstract:	The function tor_addr_parse_mask_ports() in address.c allocates memory on line 585 and fails to free it.		
Sink:	address.c:585 base = _tor_strdup(...)		

```
583      goto err;
584    }
585    base = tor_strdup(s);
586
587    /* Break 'base' into separate strings. */
```

Category: String Termination Error (54 Issues)



Abstract:

The program relies on proper string termination, but an intermediate function might have caused the source buffer to become unterminated. This could result in a buffer overflow.

Explanation:

String termination errors caused by truncation occur when:

1. Data enters a program.

2. The data passes through a function that truncates it, removing the null terminator.

Examples of functions that truncate data include strncpy(), wcsncpy() and _tcsncpy().

3. The data is passed to a function that requires its input to be null terminated.

Example 1: The following code retrieves the value of a null-terminated environment variable and uses strncpy() to copy the data into new . Later, the program incorrectly assumes new will always be null terminated when it is passed to setenv().

```
...
char *value = getenv("PWD");
...
char *new_value = strncpy(new, value, strlen(value));
setenv("PATH", new, 1);
...
```

Because the call to strncpy() is bounded by the length of the string as computed by strlen(), which does not account for the null terminator, new will become unterminated and cause the call to setenv() to behave incorrectly. The function setenv() will continue copying from the memory following new until it encounters an arbitrary null character. If the function does not find a null terminator before reaching the maximum size of the program's environment, the behavior of the function and other environment functions is undefined. Even if an arbitrary null character is found, the PATH environment variable might be left pointing to invalid directories. Worse yet, if the attacker can control values in memory that follow new, then they might introduce malicious entries to PATH, thereby changing the meaning of commands executed subsequently.

Example 2: In the following code, fgets() retrieves data from a stream and stores it in buf. The function guarantees that the data will not exceed the buffer size and that the result is null terminated. Later, strncpy() is used to copy a subset of the data to a new buffer, data. The length of the resulting value is then calculated using strlen().

```
...
char buf[MAXLEN];
fgets(buf, MAXLEN, stream);
...
strncpy(data, buf, data_size);
...
int length = strlen(data);
...
```

The code in Example 2 will behave incorrectly if data_size is less than or equal to the length of data read from the stream because the null terminator will not be copied to data. In testing, vulnerabilities like this one might not be caught because the memory immediately following data will often be null, thereby causing strlen() to produce the correct answer accidentally. However, in practice, strlen() will continue traversing memory until it encounters an arbitrary null character on the stack, which can result in a value of length that is much larger than the size of buf. Subsequent operations on data that rely on length might result in buffer overflow.

Traditionally, strings are represented as a region of memory containing data terminated with a null character. Older string-handling methods frequently rely on this null character to determine the length of the string. If a string is truncated by a function that does not guarantee null termination, the length function will read past the end of the buffer.

Malicious users could exploit this type of vulnerability by injecting data with an unexpectedly large size into the application. They may provide the malicious input either directly as input to the program or indirectly by modifying application resources, such as configuration files. In the event that an attacker causes the application to read beyond the bounds of a buffer, the attacker may be able to use a resulting buffer overflow to inject and execute arbitrary code on the system.

Recommendations:

If the program is written for Windows(R), replace all calls to strlen() with the strsafe.h equivalents StringCbLength(), which takes a maximum buffer size in bytes, or StringCchLength(), which takes a maximum buffer size in characters. If the length of the provided string is greater than the specified buffer size, these functions return an error; otherwise they behave in the same way as strlen().

If the program is written for a platform where these bounded replacements are not available, consider implementing a proprietary bounded equivalent to strlen() that behaves in the same way as the strsafe.h functions described above. If strlen() must be used, perform careful manual null termination of strings before they are passed to strlen() and audit occurrences of strlen() to ensure the correct usage. You can use a custom rule to unconditionally flag this function for audit. See the custom rule writing section of the Rules Builder documentation for more information.

address.c, line 1037 (String Termination Error)

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

Abstract: The function tor_addr_parse() in address.c relies on proper string termination when it calls strlen() on line 1037, but an intermediate copy function might have truncated the source buffer and caused it to become unterminated. Relying on proper null termination could result in buffer overflow.

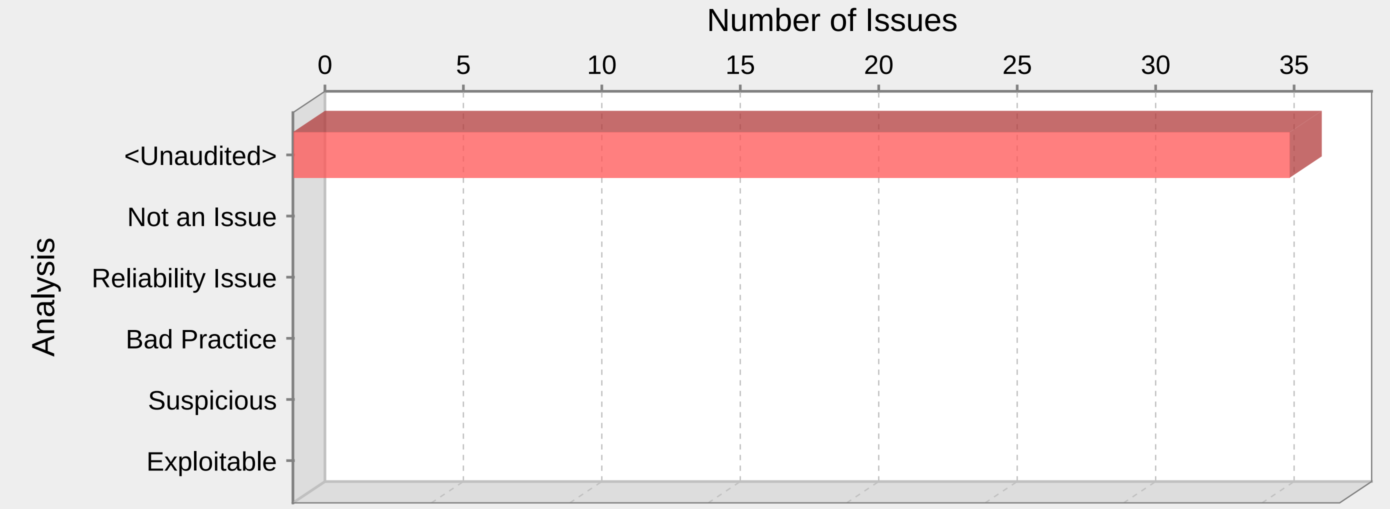
Source: util.c:4383 fgets()

```
4381      tor_assert(count <= INT_MAX);
4382
4383      retval = fgets(buf_out, (int)count, stream);
4384
4385      if (!retval) {
```

Sink: address.c:1037 strlen()

```
1035      tor_assert(addr && src);
1036      if (src[0] == '[' && src[1])
1037        src = tmp = tor_strndup(src+1, strlen(src)-2);
1038
1039      if (tor_inet_pton(AF_INET6, src, &in6_tmp) > 0) {
```

Category: Buffer Overflow (36 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. Replacing less secure functions like memcpy() with their more secure versions, such as memcpy_s(), 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.

compat.c, line 374 (Buffer Overflow)

Fortify Priority:	High	**Folder**	High
Kingdom:	Input Validation and Representation		
Abstract:	The function tor_vsnprintf() in compat.c might be able to write outside the bounds of allocated memory on line 374, which could corrupt data, cause the program to crash, or lead to the execution of malicious code.		

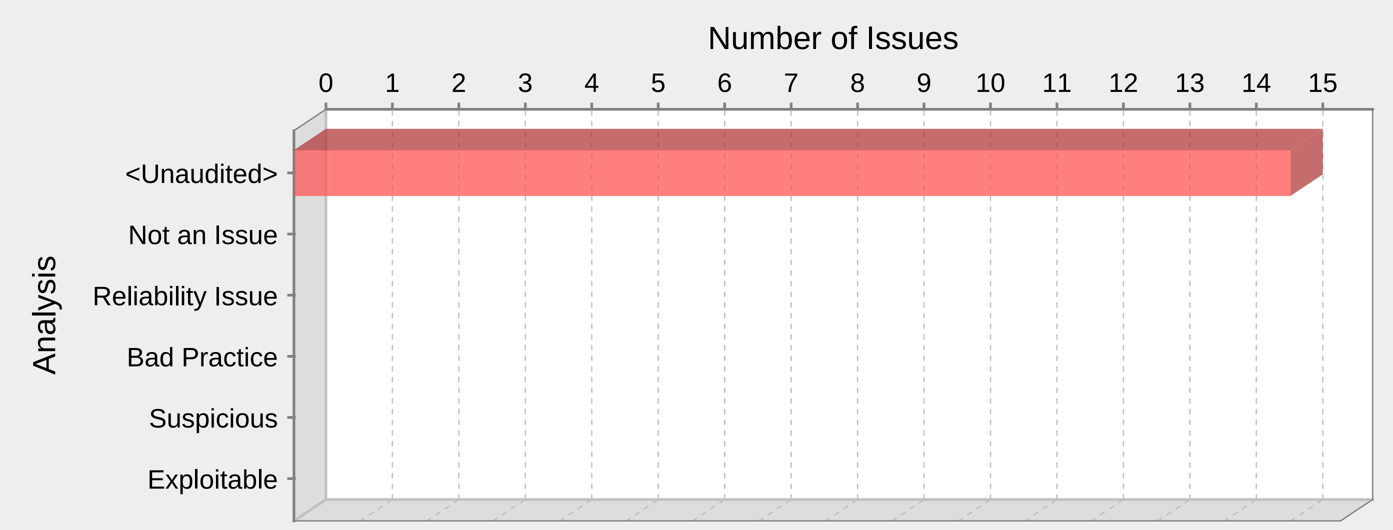
Source: util.c:2901 readdir()

```
2899
2900          result = smartlist_new();
2901          while ((de = readdir(d))) {
2902            if (!strcmp(de->d_name, ".") ||
2903                !strcmp(de->d_name, ".."))
```

Sink: compat.c:374 Assignment to str[]()

```
  r = vsnprintf(str, size, format, args);
#endif
  str[size-1] = '\0';
  if (r < 0 || r >= (ssize_t)size)
    return -1;
```

Category: Race Condition: File System Access (15 Issues)



Abstract:

The window of time between when a file property is checked and when the file is used can be exploited to launch a privilege escalation attack.

Explanation:

File access race conditions, known as time-of-check, time-of-use (TOCTOU) race conditions, occur when:

1. The program checks a property of a file, referencing the file by name.

2. The program later performs a filesystem operation using the same filename and assumes that the previously-checked property still holds.

Example 1: The following code is from a program installed setuid root. The program performs certain file operations on behalf of non-privileged users, and uses access checks to ensure that it does not use its root privileges to perform operations that should otherwise be unavailable the current user. The program uses the access() system call to check if the person running the program has permission to access the specified file before it opens the file and performs the necessary operations.

```
if (!access(file,W_OK)) {
f = fopen(file,"w+");
operate(f);
...
}
else {
fprintf(stderr,"Unable to open file %s.\n",file);
}
```

The call to access() behaves as expected, and returns 0 if the user running the program has the necessary permissions to write to the file, and -1 otherwise. However, because both access() and fopen() operate on filenames rather than on file handles, there is no guarantee that the file variable still refers to the same file on disk when it is passed to fopen() that it did when it was passed to access(). If an attacker replaces file after the call to access() with a symbolic link to a different file, the program will use its root privileges to operate on the file even if it is a file that the attacker would otherwise be unable to modify. By tricking the program into performing an operation that would otherwise be impermissible, the attacker has gained elevated privileges.

This type of vulnerability is not limited to programs with root privileges. If the application is capable of performing any operation that the attacker would not otherwise be allowed perform, then it is a possible target.

The window of vulnerability for such an attack is the period of time between when the property is tested and when the file is used. Even if the use immediately follows the check, modern operating systems offer no guarantee about the amount of code that will be executed before the process yields the CPU. Attackers have a variety of techniques for expanding the length of the window of opportunity in order to make exploits easier, but even with a small window, an exploit attempt can simply be repeated over and over until it is successful.

Example 2: The following code creates a file and then changes the owner of the file.

```
fd = creat(FILE, 0644);  /* Create file */
if (fd == -1)
return;
if (chown(FILE, UID, -1) < 0) {  /* Change file owner */
...
}
```

The code assumes that the file operated upon by the call to chown() is the same as the file created by the call to creat(), but that is not necessarily the case. Because chown() operates on a file name and not on a file handle, an attacker might be able to replace the file with a link to file the attacker does not own. The call to chown() would then give the attacker ownership of the linked file.

Recommendations:

To prevent file access race conditions, you must ensure that a file cannot be replaced or modified once the program has begun a series of operations on it. Avoid functions that operate on filenames, since they are not guaranteed to refer to the same file on disk outside of the scope of a single function call. Open the file first and then use functions that operate on file handles rather than filenames.

The most effective way to check file access permissions is to drop to the privilege of the current user and attempt to open the file with those reduced privileges. If the file open succeeds, additional access checks can be performed atomically using the resulting file handle. If the file open fails, then the user does not have access to the file and the operation should be aborted. By dropping to the user's privilege before attempting a series of file operations, the program cannot be easily tricked by changes to the underlying filesystem.

Tips:

1. Be careful, a race condition can still exist after the file is opened if later operations depend on a property that was checked before the file was opened. For example, if a stat structure is populated before a file is opened, and then a later decision about whether to operate on the file is based on a value read from the stat structure, the file could be modified prior to being opened, rendering the stat information stale. Always verify that file operations are performed on open file handles rather than filenames.

crypto.c, line 1893 (Race Condition: File System Access)

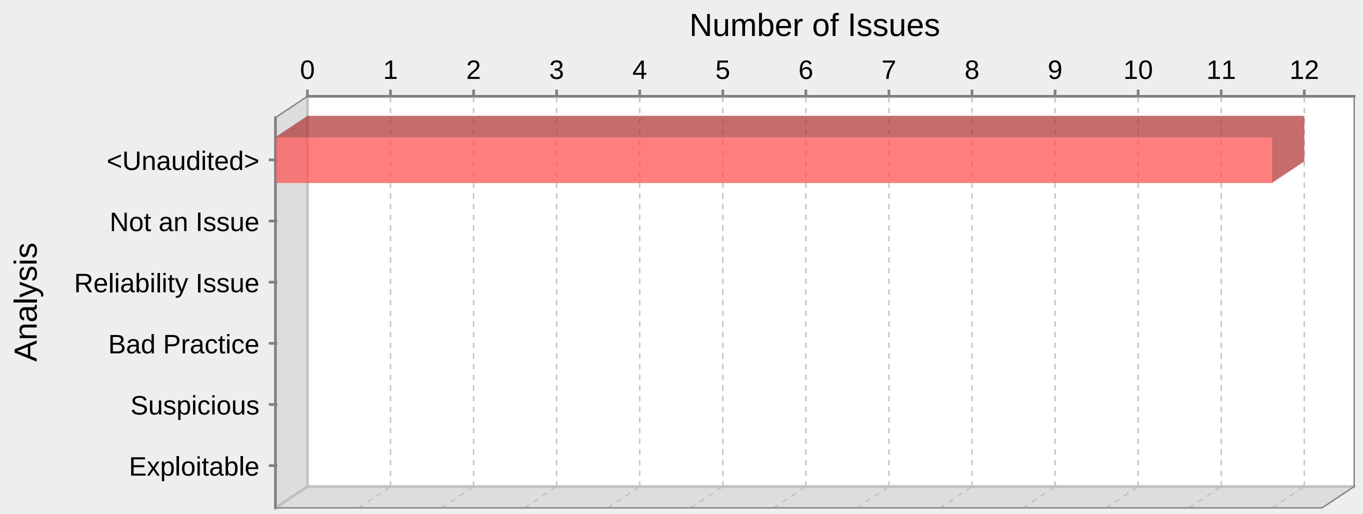
Fortify Priority:	High	**Folder**	High
Kingdom:	Time and State		

Abstract: The window of time between the call to <a href="location://common/util.c###2278###2278###0###0">read_file_to_str()</a> and <a href="location://common/compat.c###720###720###0###0">replace_file()</a> can be exploited to launch a privilege escalation attack.

Sink: crypto.c:1893 replace_file(fname, ?) : Symbolic filename fname used to operate on a file()

```
1891            log_warn(LD_CRYPTO, "Moving broken dynamic DH prime to '%s'.", fname_new);
1892
1893            if (replace_file(fname, fname_new))
1894              log_notice(LD_CRYPTO, "Error while moving '%s' to '%s'.",
1895                         fname, fname_new);
```

Category: Null Dereference (12 Issues)



Abstract:

The program can potentially dereference a null pointer, thereby causing a segmentation fault.

Explanation:

Null pointer exceptions usually occur when one or more of the programmer's assumptions is violated. A dereference-after-store error occurs when a program explicitly sets a pointer to null and dereferences it later. This error is often the result of a programmer initializing a variable to null when it is declared.

Most null pointer issues result in general software reliability problems, but if an attacker can intentionally trigger a null pointer dereference, the attacker might be able to use the resulting exception to bypass security logic or to cause the application to reveal debugging information that will be valuable in planning subsequent attacks.

Example: In the following code, the programmer explicitly sets the variable ptr to NULL. Later, the programmer dereferences ptr before checking the object for a null value.

```
*ptr = NULL;
...
ptr->field = val;
...
}
```

Recommendations:

Implement careful checks before dereferencing objects that might be null. When possible, abstract null checks into wrappers around code that manipulates resources to ensure that they are applied in all cases and to minimize the places where mistakes can occur.

circuitlist.c, line 1487 (Null Dereference)

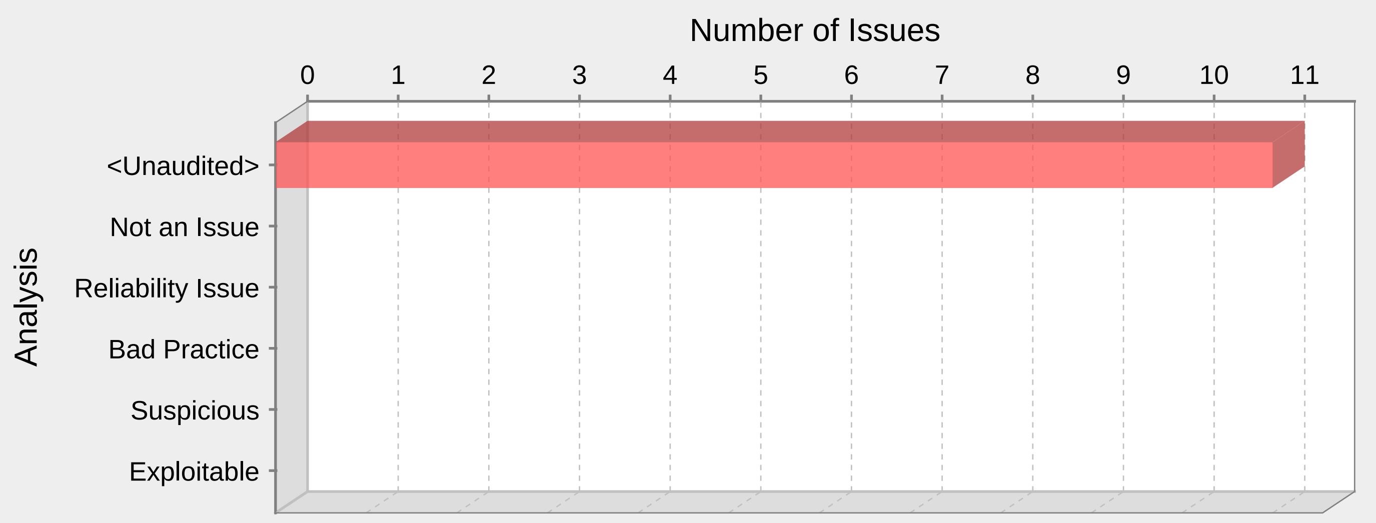
Fortify Priority:	High	**Folder**	High
Kingdom:	Code Quality		

Abstract: The function assert_circuit_ok() in circuitlist.c can crash the program by dereferencing a null pointer on line 1487.

Sink: circuitlist.c:1487 Dereferenced : or_circ()

```
1485        tor_assert(or_circ);
1486        if (!c->marked_for_close) {
1487          tor_assert(or_circ->rend_splice);
1488          tor_assert(or_circ->rend_splice->rend_splice == or_circ);
1489        }
```

Category: Key Management: Hardcoded Encryption Key (11 Issues)



Abstract:

Encryption keys can compromise system security in a way that cannot be easily remedied.

Explanation:

It is never a good idea to hardcode an encryption key. Not only does hardcoding an encryption key allow all of the project's developers to view the encryption key, it also makes fixing the problem extremely difficult. Once the code is in production, the encryption key cannot be changed without patching the software. If the account protected by the encryption key is compromised, the owners of the system will be forced to choose between security and availability.

Example: The following code uses a hardcoded encryption key:

```
...
char encyrptionKey[] = "lakdsljkalkjlksdfkl";
...
```

Anyone who has access to the code will have access to the encryption key. Once the program has shipped, there is no way to change the encryption key unless the program is patched. An employee with access to this information can use it to break into the system. Even worse, if attackers have access to the executable for the application they can disassemble the code, which will contain the value of the encryption key used.

Recommendations:

Encryption keys should never be hardcoded and should generally be obfuscated and managed in an external source. Storing encryption keys in plaintext anywhere on the system allows anyone with sufficient permissions to read and potentially misuse the encryption key.

config.c, line 6823 (Key Management: Hardcoded Encryption Key)

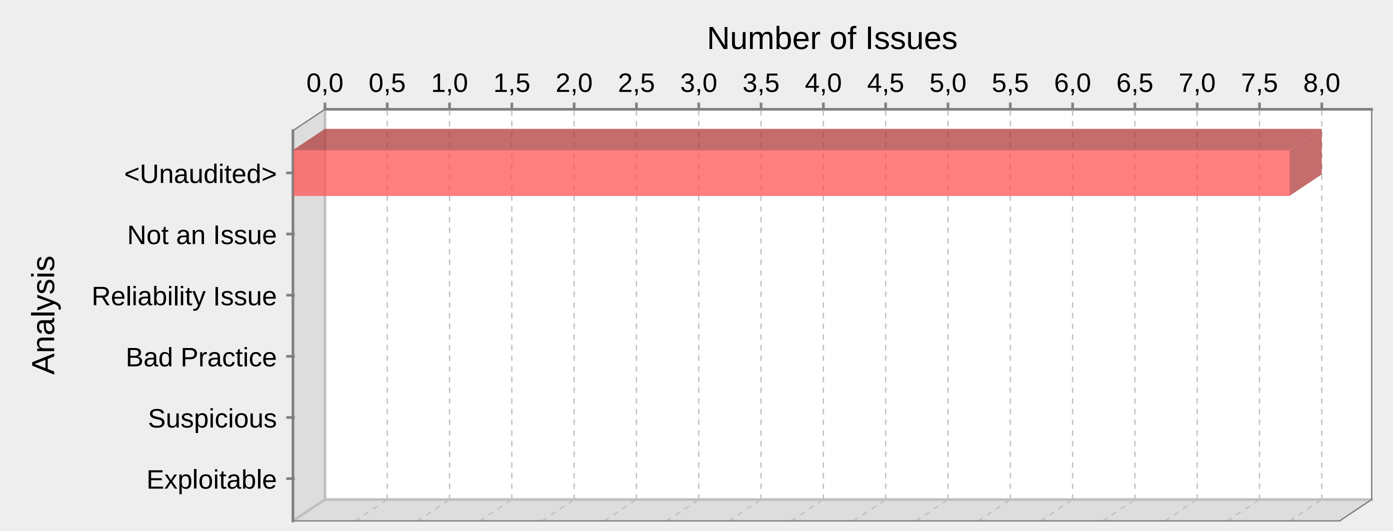
Fortify Priority:	High	**Folder**	High
Kingdom:	Security Features		

Abstract: Encryption keys can compromise system security in a way that cannot be easily remedied.

Sink: config.c:6823 FunctionCall: strcmp()

```
6821
6822      for (line = state->TransportProxies ; line ; line = line->next) {
6823        tor_assert(!strcmp(line->key, "TransportProxy"));
6824        if (!state_transport_line_is_valid(line->value))
6825          broken = 1;
```

Category: Integer Overflow (8 Issues)



Abstract:

Not accounting for integer overflow can result in logic errors or buffer overflow.

Explanation:

Integer overflow errors occur when a program fails to account for the fact that an arithmetic operation can result in a quantity either greater than a data type's maximum value or less than its minimum value. These errors often cause problems in memory allocation functions, where user input intersects with an implicit conversion between signed and unsigned values. If an attacker can cause the program to under-allocate memory or interpret a signed value as an unsigned value in a memory operation, the program may be vulnerable to a buffer overflow.

Example 1: The following code excerpt from OpenSSH 3.3 demonstrates a classic case of integer overflow:

```
nresp = packet_get_int();
if (nresp > 0) {
response = xmalloc(nresp*sizeof(char*));
for (i = 0; i < nresp; i++)
response[i] = packet_get_string(NULL);
}
```

If nresp has the value 1073741824 and sizeof(char*) has its typical value of 4, then the result of the operation nresp*sizeof(char*) overflows, and the argument to xmalloc() will be 0. Most malloc() implementations will happily allocate a 0-byte buffer, causing the subsequent loop iterations to overflow the heap buffer response.

Example 2: This example processes user input comprised of a series of variable-length structures. The first 2 bytes of input dictate the size of the structure to be processed.

```
char* processNext(char* strm) {
char buf[512];
short len = *(short*) strm;
strm += sizeof(len);
if (len <= 512) {
memcpy(buf, strm, len);
process(buf);
return strm + len;
} else {
return -1;
}
}
```

The programmer has set an upper bound on the structure size: if it is larger than 512, the input will not be processed. The problem is that len is a signed integer, so the check against the maximum structure length is done with signed integers, but len is converted to an unsigned integer for the call to memcpy(). If len is negative, then it will appear that the structure has an appropriate size (the if branch will be taken), but the amount of memory copied by memcpy() will be quite large, and the attacker will be able to overflow the stack with data in strm.

Recommendations:

There are no simple guidelines that allow you to avoid every integer overflow problem, but these recommendations will help prevent the most egregious cases:

- Pay attention to compiler warnings related to signed/unsigned conversions. Some programmers may believe that these warnings are innocuous, but they sometimes point out potential integer overflow problems.

- Be vigilant about checking reasonable upper and lower bounds for all program input. Even if the program should only be dealing with positive integers, check to be sure that the values you are processing are not less than zero. (You can eliminate the need for a lower bounds check by using unsigned data types.)

- Be conservative about the range of values you allow.

- Be cognizant of the implicit typecasting that takes place when you call functions,

perform arithmetic operations or compare values of different types.

Example 3: The code below implements a wrapper function designed to allocate memory for an array safely by performing an appropriate check on its arguments prior to making a call to malloc().

```
void* arrmalloc(uint sz, uint nelem) {
void *p;
if(sz > 0 && nelem >= UINT_MAX / sz)
return 0;
return malloc(sz * nelem);
}
```

Tips:

1. Consider whether or not the integer that might overflow has been derived from the length of a string. If it has, it is safe to assume that the string fits inside the process's address space, which puts an upper bound on the string's length. This fact alone does not make integer overflow impossible, but it does put additional constraints on the arithmetic that must be performed in order to cause an overflow.

util.c, line 144 (Integer Overflow)

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

Abstract: The function _tor_malloc() in util.c does not account for integer overflow, which can result in a logic error or a buffer overflow.

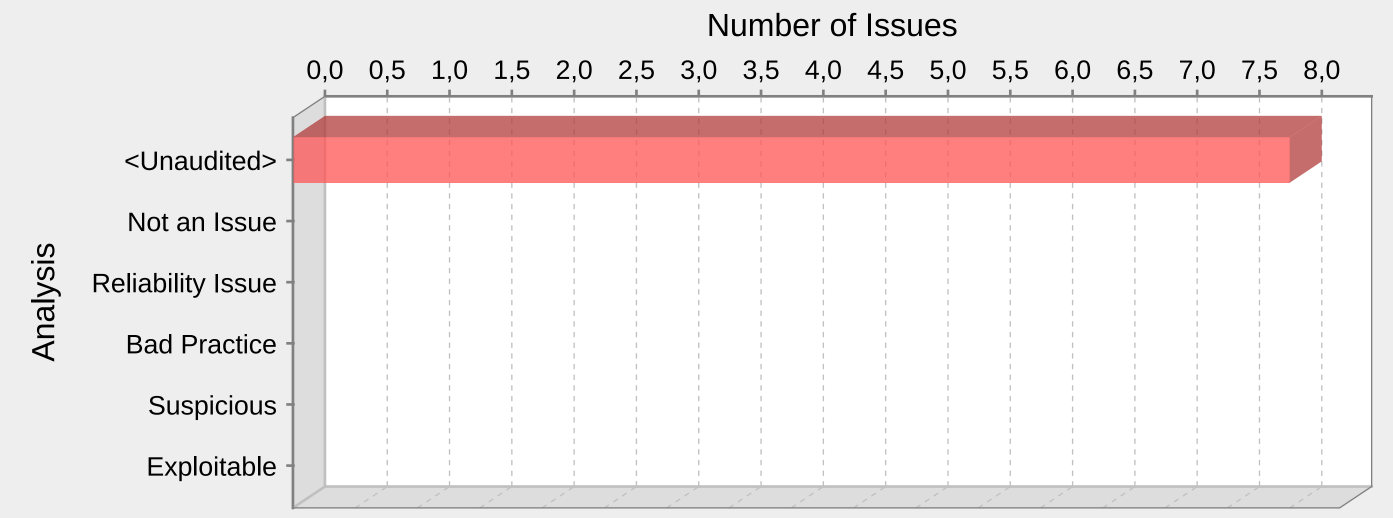
Source: compat.c:200 mmap()

```
198       }
199
200       string = mmap(0, size, PROT_READ, MAP_PRIVATE, fd, 0);
201       close(fd);
202       if (string == MAP_FAILED) {
```

Sink: util.c:144 malloc()

```
142       result = dmalloc_malloc(file, line, size, DMALLOC_FUNC_MALLOC, 0, 0);
143     #else
144       result = malloc(size);
145     #endif
```

Category: Often Misused: Privilege Management (8 Issues)



Abstract:

Failure to adhere to the principle of least privilege amplifies the risk posed by other vulnerabilities.

Explanation:

Programs that run with root privileges have caused innumerable Unix security disasters. It is imperative that you carefully review privileged programs for all kinds of security problems, but it is equally important that privileged programs drop back to an unprivileged state as quickly as possible in order to limit the amount of damage that an overlooked vulnerability might be able to cause.

Privilege management functions can behave in some less-than-obvious ways, and they have different quirks on different platforms. These inconsistencies are particularly pronounced if you are transitioning from one non-root user to another.

Signal handlers and spawned processes run at the privilege of the owning process, so if a process is running as root when a signal fires or a sub-process is executed, the signal handler or sub-process will operate with root privileges. An attacker may be able to leverage these elevated privileges to do further damage.

Recommendations:

If a program can be rewritten so that it does not need root access, rewrite it.

Disable signals before elevating privileges to avoid having signal handling code run with unexpected privileges. Re-enable signals after dropping back to user privilege.

Whenever possible, drop privileges by calling setuid() with a non-zero argument immediately after completing privileged operations.

In some situations it may be impossible for an application to make use of the setuid()/setgid() calls to drop privileges. This usually occurs when the application needs the ongoing ability to perform some operation as root on behalf of the user. For example, an FTP daemon needs to bind to ports between 1 and 1024 in order to service user requests. In such cases, seteuid() and getegid() should be used to temporarily drop to a lower effective privilege level while retaining the ability to return to root privileges when necessary. Note that it may be equally easy for an attacker to return the application to root privileges as well, so only use seteuid()/getegid() when there is no way to drop privileges entirely.

The following program gives a general outline of a well-constructed setuid root program:

```
int main(int argc, char** argv) {
uid_t runner_uid = getuid();
uid_t runner_gid = getgid();
uid_t owner_uid = geteuid();
uid_t owner_gid = getegid();
int sigmask;

/* Drop privileges right up front, but
we'll need them back in a little bit,
so use effective id */
if (setreuid(owner_uid, runner_uid) ||
setregid(owner_gid, runner_gid)) {
exit(-1);
}

/* privilege not necessary or desirable at this point */
processCommandLine(argc, argv);

/* disable signal handling */
sigmask = sigprocmask(~0);

/* Take privileges back */
if (setreuid(runner_uid, owner_uid) ||
setregid(runner_gid, owner_gid)) {
exit(-1);
}

openSocket(88); /* requires root */

/* Drop privileges for good */
if (setuid(runner_uid) || setgid(runner_gid)) {
exit(-1);
}

/* re-enable signals */
sigprocmask(sigmask);

doWork();
}
```

Another approach to retaining some privileges while minimizing risk is to partition the program into privileged and unprivileged pieces. Create two processes: one that does the majority of the work and runs without privileges, and a second that retains privileges but only carries out a very limited number of operations at the request of the other process.

Chen and Wagner offer up wrapper functions that provide a consistent and easily-understood interface for privilege management [1].

Tips:

1. Review the code surrounding this call. Determine the extent of the code that will be executed with elevated privileges. Which operations absolutely require elevated privileges? Is there a way to reduce the amount of code in the privileged section?

2. Be sure that the program checks the return value of any privilege management functions it invokes. If attackers can prevent a privilege transition from taking place, they may be able to take advantage of the fact that the program is executing under unexpected conditions.

compat.c, line 1524 (Often Misused: Privilege Management)

Fortify Priority:	High	**Folder**	High
Kingdom:	API Abuse		

Abstract: The function switch_id() in compat.c fails to adhere to the principle of least privilege, which greatly amplifies the risk posed by other vulnerabilities.

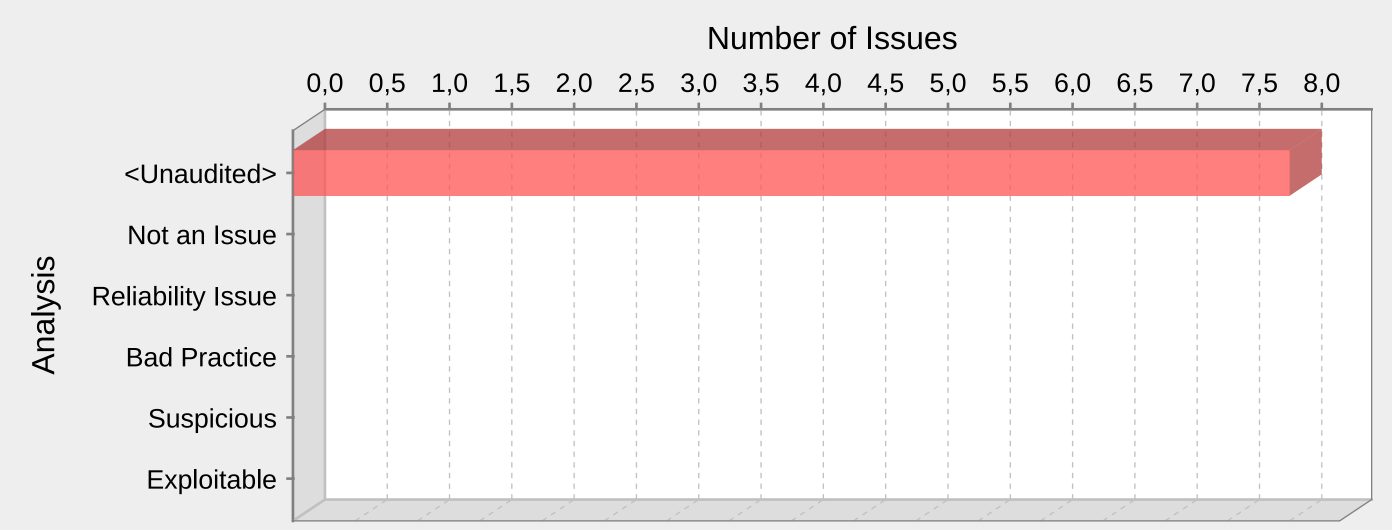
Source: compat.c:1501 getpwnam()

```
1499
1500        /* Lookup the user and group information, if we have a problem, bail out. */
1501        pw = getpwnam(user);
1502        if (pw == NULL) {
1503          log_warn(LD_CONFIG, "Error setting configured user: %s not found", user);
```

Sink: compat.c:1524 setegid()

```
1522        }
1523
1524        if (setegid(pw->pw_gid)) {
1525          log_warn(LD_GENERAL, "Error setting egid to %d: %s",
1526                   (int)pw->pw_gid, strerror(errno));
```

Category: Path Manipulation (8 Issues)



Abstract:

Allowing user input to control paths used in filesystem operations could enable an attacker to access or modify otherwise protected system resources.

Explanation:

Path manipulation errors occur when the following two conditions are met:

1. An attacker can specify a path used in an operation on the filesystem.

2. By specifying the resource, the attacker gains a capability that would not otherwise be permitted.

For example, the program may give the attacker the ability to overwrite the specified file or run with a configuration controlled by the attacker.

Example 1: The following code uses input from a CGI request to create a file name. The programmer has not considered the possibility that an attacker could provide a file name such as "../../apache/conf/httpd.conf", which will cause the application to delete the specified configuration file.

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

Example 2: The following code uses input from the command line to determine which file to open and echo back to the user. If the program runs with privileges and malicious users can create soft links to the file, they can use the program to read the first part of any file on the system.

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

Recommendations:

The best way to prevent path manipulation is with a level of indirection: create a list of legitimate resource names that a user is allowed to specify, and only allow the user to select from the list. With this approach the input provided by the user is never used directly to specify the resource name.

In some situations this approach is impractical because the set of legitimate resource names is too large or too hard to keep track of. Programmers often resort to blacklisting in these situations. Blacklisting selectively rejects or escapes potentially dangerous characters before using the input. However, any such list of unsafe characters is likely to be incomplete and will almost certainly become out of date. A better approach is to create a white list of characters that are allowed to appear in the resource name and accept input composed exclusively of characters in the approved set.

Tips:

1. If the program is performing input validation, satisfy yourself that the validation is correct, and use the Custom Rules Editor to create a cleanse rule for the validation routine.

2. It is notoriously difficult to correctly implement a blacklist. If the validation logic relies on blacklisting, be skeptical. Consider different types of input encoding and different sets of meta-characters that might have special meaning when interpreted by different operating systems, databases, or other resources. Determine whether or not the blacklist can be updated easily, correctly, and completely if these requirements ever change.

compat.c, line 128 (Path Manipulation)

Fortify Priority:	Critical	**Folder**	Critical
Kingdom:	Input Validation and Representation		
Abstract:	Attackers can control the filesystem path argument to open() at compat.c line 128, which allows them to access or modify otherwise protected files.		

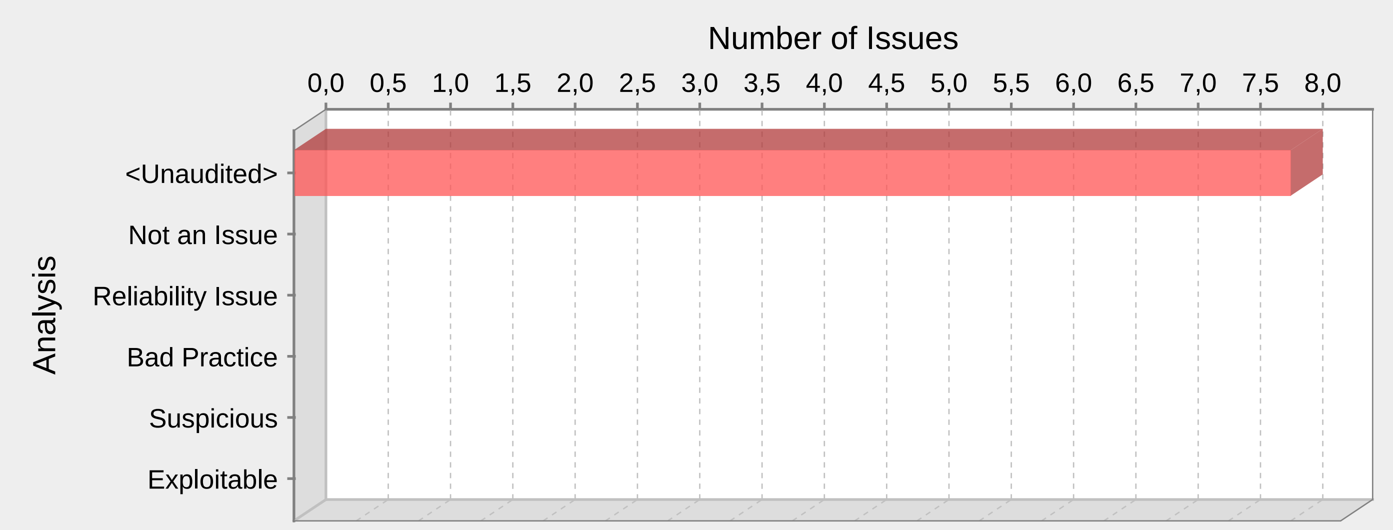
Source: tor-checkkey.c:18 main(1)

```
16
17          int
18          main(int c, char **v)
19          {
20            crypto_pk_t *env;
```

Sink: compat.c:128 open()

```
126           int fd;
127         #ifdef O_CLOEXEC
128           fd = open(path, flags|O_CLOEXEC, mode);
129           if (fd >= 0)
130             return fd;
```

Category: Unreleased Resource (8 Issues)



Abstract:

The program can potentially fail to release a system resource.

Explanation:

The program can potentially fail to release a system resource.

Resource leaks have at least two common causes:

- Error conditions and other exceptional circumstances.

- Confusion over which part of the program is responsible for releasing the resource.

Most unreleased resource issues result in general software reliability problems, but if an attacker can intentionally trigger a resource leak, the attacker might be able to launch a denial of service by depleting the resource pool.

Example: The following function does not close the file handle it opens if an error occurs. If the process is long-lived, the process can run out of file handles.

```
int decodeFile(char* fName)
{
char buf[BUF_SZ];
FILE* f = fopen(fName, "r");

if (!f) {
printf("cannot open %s\n", fName);
return DECODE_FAIL;
} else {
while (fgets(buf, BUF_SZ, f)) {
if (!checkChecksum(buf)) {
return DECODE_FAIL;
} else {
decodeBlock(buf);
}
}
}
fclose(f);
return DECODE_SUCCESS;
}
```

Recommendations:

Because resource leaks can be hard to track down, establish a set of resource management patterns and idioms for your software and do not tolerate deviations from your conventions.

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

```
int decodeFile(char* fName)
```

```
{
char buf[BUF_SZ];
FILE* f = fopen(fName, "r");
if (!f) {
goto ERR;
} else {
while (fgets(buf, BUF_SZ, f)) {
if (!checkChecksum(buf)) {
goto ERR;
} else {
decodeBlock(buf);
}
}
}
fclose(f);
return DECODE_SUCCESS;
ERR:
if (!f) {
printf("cannot open %s\n", fName);
} else {
fclose(f);
}
return DECODE_FAIL;
}
```

compat.c, line 2878 (Unreleased Resource)

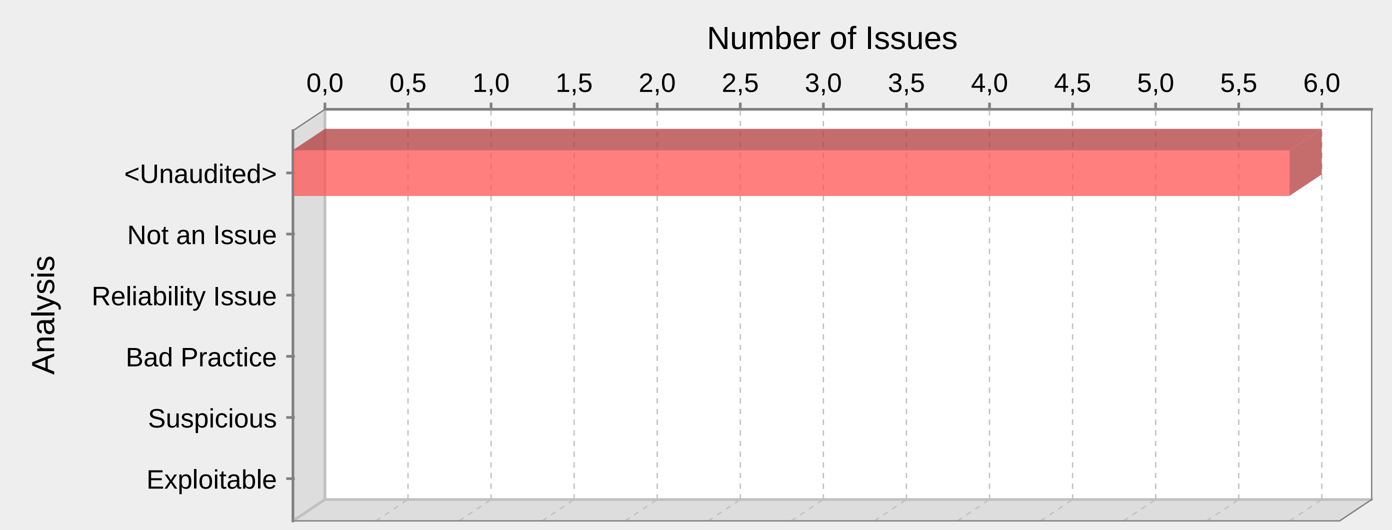
Fortify Priority:	High	**Folder**	High
Kingdom:	Code Quality		

Abstract: The function tor_mlockall() in compat.c sometimes fails to release a system resource allocated by <a href="location:///usr/include/sys/mman.h###112###112###0###0">mlockall()</a> on line 2878.

Sink: compat.c:2878 mlockall(...) : Resource allocated()

```
2876      }
2877
2878      if (mlockall(MCL_CURRENT|MCL_FUTURE) == 0) {
2879        log_info(LD_GENERAL, "Insecure OS paging is effectively disabled.");
2880        return 0;
```

Category: Race Condition: Signal Handling (6 Issues)



Abstract:

Installing the same signal handler for multiple signals can lead to a race condition when different signals are caught in short succession.

Explanation:

Signal handling race conditions can occur whenever a function installed as a signal handler is non-reentrant, which means it maintains some internal state or calls another function that does so. Such race conditions are even more likely when the same function is installed to handle multiple signals.

Signal handling race conditions are more likely to occur when:

1. The program installs a single signal handler for more than one signal.

2. Two different signals for which the handler is installed arrive in short succession, causing a race condition in the signal handler.

Example: The following code installs the same simple, non-reentrant signal handler for two different signals. If an attacker causes signals to be sent at the right moments, the signal handler will experience a double free vulnerability. Calling free() twice on the same value can lead to a buffer overflow. When a program calls free() twice with the same argument, the program's memory management data structures become corrupted. This corruption can cause the program to crash or, in some circumstances, cause two later calls to malloc() to return the same pointer. If malloc() returns the same value twice and the program later gives the attacker control over the data that is written into this doubly-allocated memory, the program becomes vulnerable to a buffer overflow attack.

```
void sh(int dummy) {
...
free(global2);
free(global1);
...
}
int main(int argc,char* argv[]) {
...
signal(SIGHUP,sh);
signal(SIGTERM,sh);
...
}
```

Recommendations:

To prevent signal handling race conditions, functions installed as signal handlers must be fully reentrant: The functions cannot maintain any internal state or call other functions that do so. Minimalism is the best rule when writing signal handlers. Perform only the bare-minimum functionality inside a signal handler and be very cautious what functions you invoke. Various lists of reentrant functions that are safe to call from a signal handler have been compiled. The POSIX standard lists the following common functions as safe to call from a signal handler:

_Exit() _exit() abort() accept() access() aio_error() aio_return() aio_suspend() alarm() bind() cfgetispeed() cfgetospeed() cfsetispeed() cfsetospeed() chdir() chmod() chown() clock_gettime() close() connect() creat() dup() dup2() execle() execve() fchmod() fchown() fcntl() fdatasync() fork() fpathconf() fstat() fsync() ftruncate() getegid() geteuid() getgid() getgroups() getpeername() getpgrp() getpid() getppid() getsockname() getsockopt() getuid() kill() link() listen() lseek() lstat() mkdir() mkfifo() open() pathconf() pause() pipe() poll() posix_trace_event() pselect() raise() read() readlink() recv() recvfrom() recvmsg() rename() rmdir() select() sem_post() send() sendmsg() sendto() setgid() setpgid() setsid() setsockopt() setuid() shutdown() sigaction() sigaddset() sigdelset() sigemptyset() sigfillset() sigismember() signal() sigpause() sigpending() sigprocmask() sigqueue() sigset() sigsuspend() sleep() socket() socketpair() stat() symlink() sysconf() tcdrain() tcflow() tcflush() tcgetattr() tcgetpgrp() tcsendbreak() tcsetattr() tcsetpgrp() time() timer_getoverrun() timer_gettime() timer_settime() times() umask() uname() unlink() utime() wait() waitpid() write()

Signal handlers are hard to get right and there is no silver-bullet advice for avoiding problems. When in doubt about whether a function is reentrant and safe to call in a signal handler, take the conservative approach and don't call the function. In the same spirit, do not install the same signal handler for multiple signals; it will increase the likelihood of a race condition.

main.c, line 2244 (Race Condition: Signal Handling)

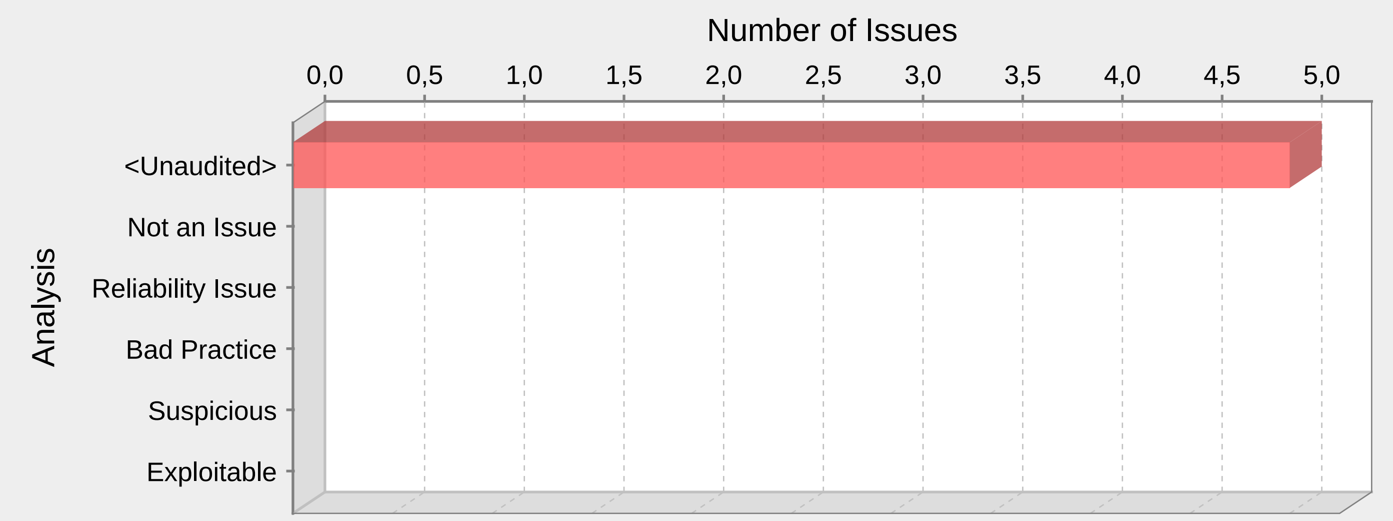
Fortify Priority:	High	**Folder**	High
Kingdom:	Time and State		

Abstract: The function handle_signals() in main.c installs the same signal handler for multiple functions, which can lead to a race condition when different signals are caught in short succession.

Sink: main.c:2244 sigaction(15, (&action), ...) : Handler (&action) registered for signal <inline expression>()

```
2242        action.sa_handler = SIG_IGN;
2243        sigaction(SIGINT,  &action, NULL);
2244        sigaction(SIGTERM, &action, NULL);
2245        sigaction(SIGPIPE, &action, NULL);
2246        sigaction(SIGUSR1, &action, NULL);
```

Category: Type Mismatch: Signed to Unsigned (5 Issues)



Abstract:

The function is declared to return an unsigned number but returns a signed value.

Explanation:

It is dangerous to rely on implicit casts between signed and unsigned numbers because the result can take on an unexpected value and violate weak assumptions made elsewhere in the program.

Example: In this example, depending on the return value of accecssmainframe(), the variable amount can hold a negative value when it is returned. Because the function is declared to return an unsigned value, amount will be implicitly cast to an unsigned number.

```
unsigned int readdata () {
int amount = 0;
...
amount = accessmainframe();
...
return amount;
}
```

If the return value of accessmainframe() is -1, then the return value of readdata() will be 4,294,967,295 on a system that uses 32-bit integers.

Conversion between signed and unsigned values can lead to a variety of errors, but from a security standpoint is most commonly associated with integer overflow and buffer overflow vulnerabilities.

Recommendations:

Although unexpected conversion between signed and unsigned quantities typically creates general quality problems, depending on the assumptions that a conversion violates, it can lead to serious security risks. Pay attention to compiler warnings related to signed/unsigned conversions. Some programmers may believe that these warnings are innocuous, but in some cases they point out potential integer overflow problems.

nodelist.c, line 50 (Type Mismatch: Signed to Unsigned)

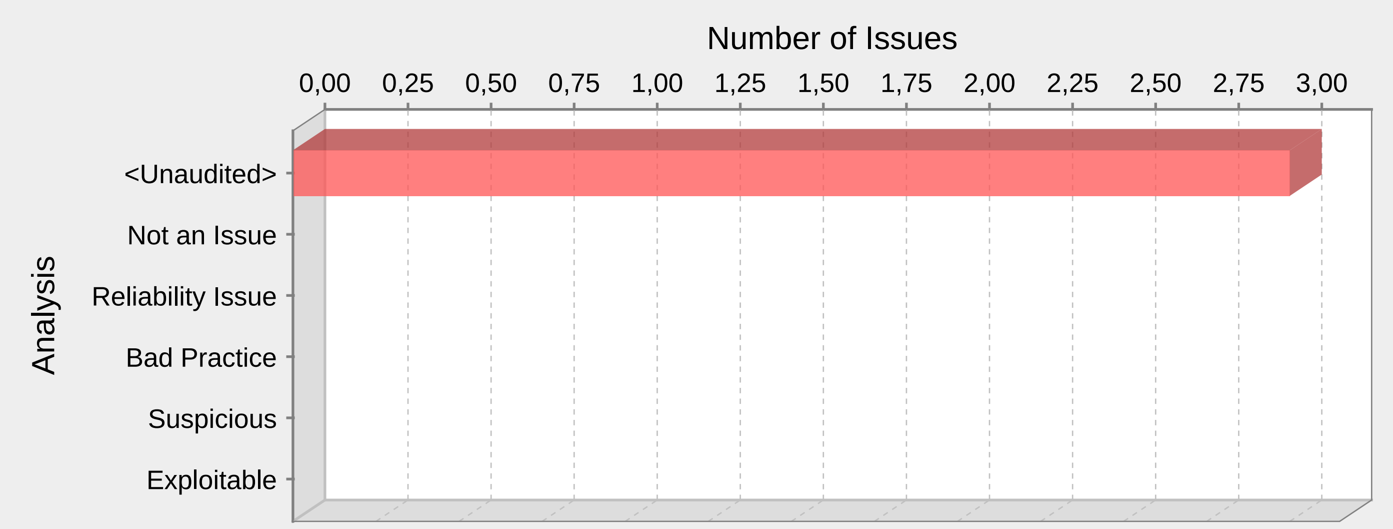
Fortify Priority:	High	**Folder**	High
Kingdom:	Code Quality		

Abstract: The function node_id_eq() in nodelist.c is declared to return an unsigned value, but on line 50 it returns a signed value.

Sink: nodelist.c:50 ReturnStatement()

```
48      node_id_eq(const node_t *node1, const node_t *node2)
49      {
50        return tor_memeq(node1->identity, node2->identity, DIGEST_LEN);
51      }
```

Category: Buffer Overflow: Off-by-One (3 Issues)



Abstract:

The program writes just past the bounds of allocated memory, which 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 off-by-one error is still common on some platforms and in some development communities, there are a variety of other types of buffer overflow, including stack and heap buffer overflows 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.

Example: The following code contains an off-by-one buffer overflow, which occurs when recv returns the maximum allowed sizeof(buf) bytes read. In this case, the subsequent dereference of buf[nbytes] will write the null byte outside the bounds of allocated memory.

```
void receive(int socket) {
char buf[MAX];
int nbytes = recv(socket, buf, sizeof(buf), 0);
buf[nbytes] = '\0';
...
}
```

Recommendations:

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. Replacing less secure functions like memcpy() with their more secure versions, such as memcpy_s(), 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.

util.c, line 1771 (Buffer Overflow: Off-by-One)

Fortify Priority:	Critical	**Folder**	Critical
Kingdom:	Input Validation and Representation		
Abstract:	The program writes just past the bounds of allocated memory, which could corrupt data, crash the program, or lead to the execution of malicious code.		

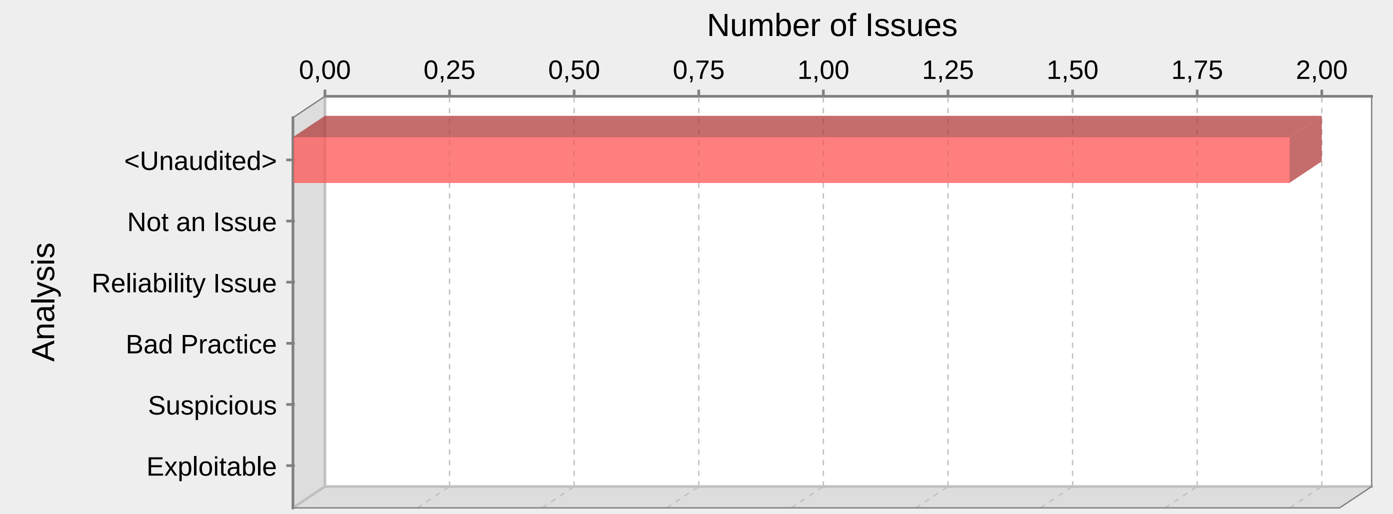
Source: util.c:1771 recv()

```
1769      while (numread != count) {
1770        if (isSocket)
1771          result = tor_socket_recv(fd, buf+numread, count-numread, 0);
1772        else
1773          result = read((int)fd, buf+numread, count-numread);
```

Sink: util.c:1771 recv()

```
1769      while (numread != count) {
1770        if (isSocket)
1771          result = tor_socket_recv(fd, buf+numread, count-numread, 0);
1772        else
1773          result = read((int)fd, buf+numread, count-numread);
```

Category: Insecure Randomness (2 Issues)



Abstract:

Standard pseudo-random number generators cannot withstand cryptographic attacks.

Explanation:

Insecure randomness errors occur when a function that can produce predictable values is used as a source of randomness in a security-sensitive context.

Computers are deterministic machines, and as such are unable to produce true randomness. Pseudo-Random Number Generators (PRNGs) approximate randomness algorithmically, starting with a seed from which subsequent values are calculated.

There are two types of PRNGs: statistical and cryptographic. Statistical PRNGs provide useful statistical properties, but their output is highly predictable and forms an easy to reproduce numeric stream that is unsuitable for use in cases where security depends on generated values being unpredictable. Cryptographic PRNGs address this problem by generating output that is more difficult to predict. For a value to be cryptographically secure, it must be impossible or highly improbable for an attacker to distinguish between it and a truly random value. In general, if a PRNG algorithm is not advertised as being cryptographically secure, it is probably a statistical PRNG and should not be used in security-sensitive contexts.

Example: The following code uses a statistical PRNG to create a URL for a receipt that remains active for some period of time after a purchase.

```
char* CreateReceiptURL() {
int num;
time_t t1;
char *URL = (char*) malloc(MAX_URL);
if (URL) {
(void) time(&t1);
srand48((long) t1); /* use time to set seed */
sprintf(URL, "%s%d%s",
"http://test.com/",lrand48(),".html");
}
return URL;
}
```

This code uses the lrand48() function to generate "unique" identifiers for the receipt pages it generates. Because lrand48() is a statistical PRNG, it is easy for an attacker to guess the strings it generates. Although the underlying design of the receipt system is also faulty, it would be more secure if it used a random number generator that did not produce predictable receipt identifiers.

Recommendations:

When unpredictability is critical, as is the case with most security-sensitive uses of randomness, use a cryptographic PRNG. Regardless of the PRNG you choose, always use a value with sufficient entropy to seed the algorithm. (Values such as the current time offer only negligible entropy and should not be used.)

There are various cross-platform solutions for C and C++ programs that offer cryptographically secure PRNGs, such as Yarrow [1], CryptLib [2], Crypt++ [3], BeeCrypt [4] and OpenSSL [5].

On Windows(R) systems, C and C++ programs can use the CryptGenRandom() function in the CryptoAPI [6]. To avoid the overhead of pulling in the entire CryptoAPI, access the underlying RtlGenRandom() function directly [7].

In the Windows .NET framework, use the GetBytes() function in any class that implements System.Security.Cryptography.RandomNumberGenerator, such as System.Security.Cryptography.RNGCryptoServiceProvider [8].

compat.c, line 2070 (Insecure Randomness)

Fortify Priority:	High	**Folder**	High
Kingdom:	Security Features		

Abstract: The random number generator implemented by srandom() cannot withstand a cryptographic attack.

Sink: compat.c:2070 srandom()

```
2068      srand(seed);
2069    #else
2070      srandom(seed);
2071    #endif
2072    }
```

Issue Count by Category

Issues by Category	
Memory Leak	1481
String Termination Error	54
Buffer Overflow	36
Illegal Pointer Value	29
Password Management: Password in Comment	28
Poor Style: Value Never Read	26
Dangerous Function: strcpy()	25
Unchecked Return Value	18
Race Condition: File System Access	15
Integer Overflow	14
Null Dereference	12
Heap Inspection	11
Key Management: Hardcoded Encryption Key	11
Redundant Null Check	11
Missing Check against Null	10
Often Misused: Privilege Management	8
Path Manipulation	8
Unreleased Resource	8
Race Condition: Signal Handling	6
Weak Encryption: Inadequate RSA Padding	6
Password Management: Null Password	5
Type Mismatch: Signed to Unsigned	5
Uninitialized Variable	5
Memory Leak: Reallocation	4
System Information Leak	4
Buffer Overflow: Off-by-One	3
Format String	3
Weak Cryptographic Hash	3
Command Injection	2
Dead Code	2
Insecure Randomness	2
Poor Style: Redundant Initialization	2
Insecure Compiler Optimization	1
Out-of-Bounds Read	1
Out-of-Bounds Read: Off-by-One	1
Portability Flaw	1
Resource Injection	1
Setting Manipulation	1

Issue Breakdown by Analysis

Issues by Analysis