tor-0.2.3.25

27/04/2013

Report Overview

Report Summary

On 27/04/2013, a source code review was performed over the tor code base. 315 files, 47.363 LOC (Executable) were scanned. A total of 1836 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

High	1620
Low	178
Critical	38

Issue Summary

Overall number of results

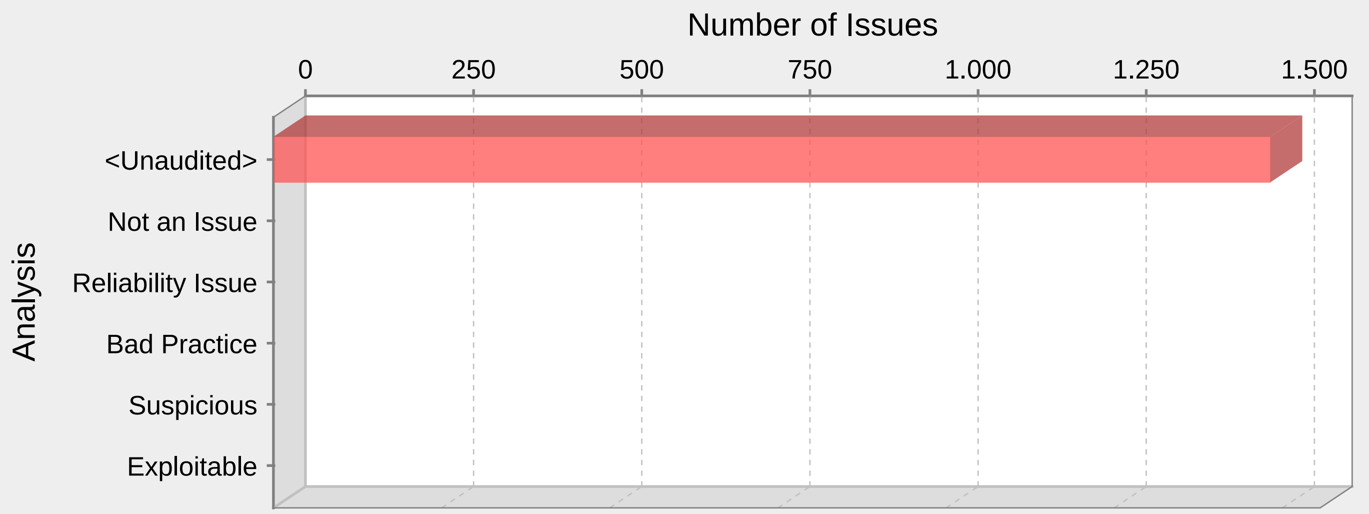
The scan found 1836 issues.

Issues by Category	
Memory Leak	1482
String Termination Error	54
Buffer Overflow	36
Illegal Pointer Value	29
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

Results Outline

Vulnerability Examples by Category

Category: Memory Leak (1482 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 1073 (Memory Leak)

Fortify Priority:	High	**Folder**	High
Kingdom:	Code Quality		
Abstract:	The function tor_addr_port_lookup() in address.c allocates memory on line 1073 and fails to free it.		
Sink:	address.c:1073 tmp = _tor_strndup(...)		

address.c, line 1308 (Memory Leak)

Fortify Priority:	High	**Folder**	High
Kingdom:	Code Quality		
Abstract:	The function get_interface_address6() in address.c allocates memory on line 1289 and fails to free it.		
Sink:	address.c:1308 smartlist_free(addrs)		

address.c, line 1426 (Memory Leak)

Fortify Priority:	High	**Folder**	High
Kingdom:	Code Quality		
Abstract:	The function addr_port_lookup() in address.c allocates memory on line 1426 and fails to free it.		
Sink:	address.c:1426 esc_addrport = esc_for_log(...)		

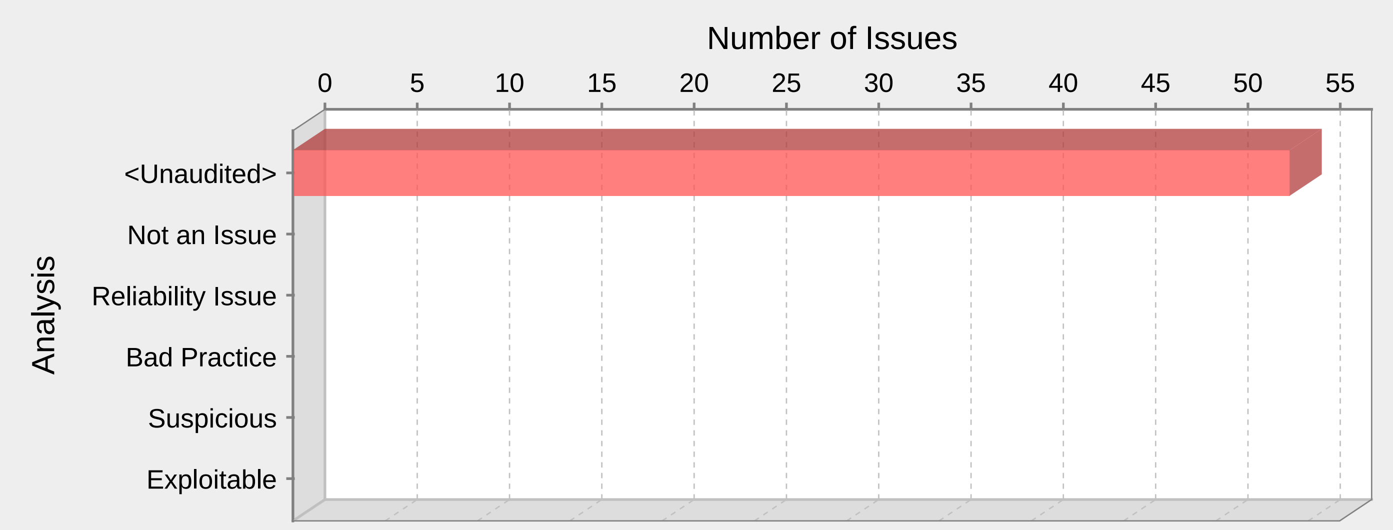
address.c, line 630 (Memory Leak)

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:630 escaped(address)		

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(...)		

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.

compat.c, line 1951 (String Termination Error)

Fortify Priority:	High	**Folder**	High
Kingdom:	Input Validation and Representation		
Abstract:	The function tor_inet_pton() in compat.c relies on proper string termination when it calls strchr() on line 1951, 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()		
Sink:	compat.c:1951 strchr()		

address.c, line 1417 (String Termination Error)

Fortify Priority:	High	**Folder**	High
Kingdom:	Input Validation and Representation		
Abstract:	The function addr_port_lookup() in address.c relies on proper string termination when it calls strrchr() on line 1417, 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()		
Sink:	address.c:1417 strrchr()		

compat.c, line 1956 (String Termination Error)

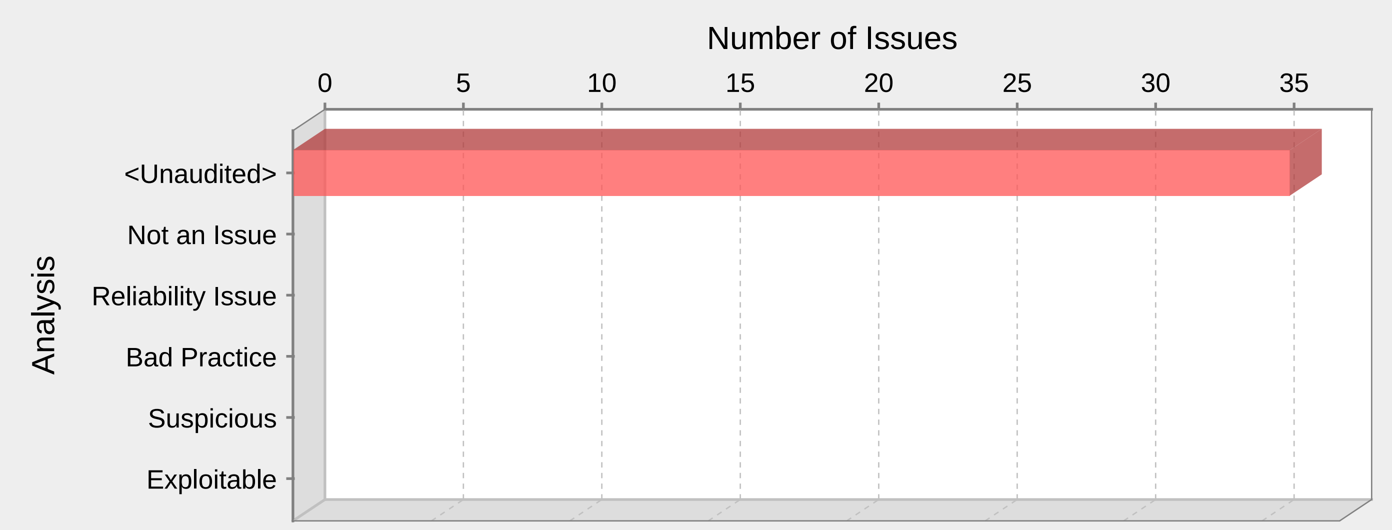
Fortify Priority:	High	**Folder**	High
Kingdom:	Input Validation and Representation		
Abstract:	The function tor_inet_pton() in compat.c relies on proper string termination when it calls strlen() on line 1956, but an intermediate copy function might have truncated the source buffer and caused it to become unterminated. Relying on proper null termination could result in buffer overflow.		
Source:	tor-resolve.c:318 main(1)		
Sink:	compat.c:1956 strlen()		

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()		
Sink:	address.c:1037 strlen()		
compat.c, line 1951 (String Termination Error)			
Fortify Priority:	High	**Folder**	High
Kingdom:	Input Validation and Representation		
Abstract:	The function tor_inet_pton() in compat.c relies on proper string termination when it calls strchr() on line 1951, 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:	tor-resolve.c:318 main(1)		
Sink:	compat.c:1951 strchr()		

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:1773 read()		
Sink:	compat.c:374 Assignment to str[]()		

compat.c, line 374 (Buffer Overflow)

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

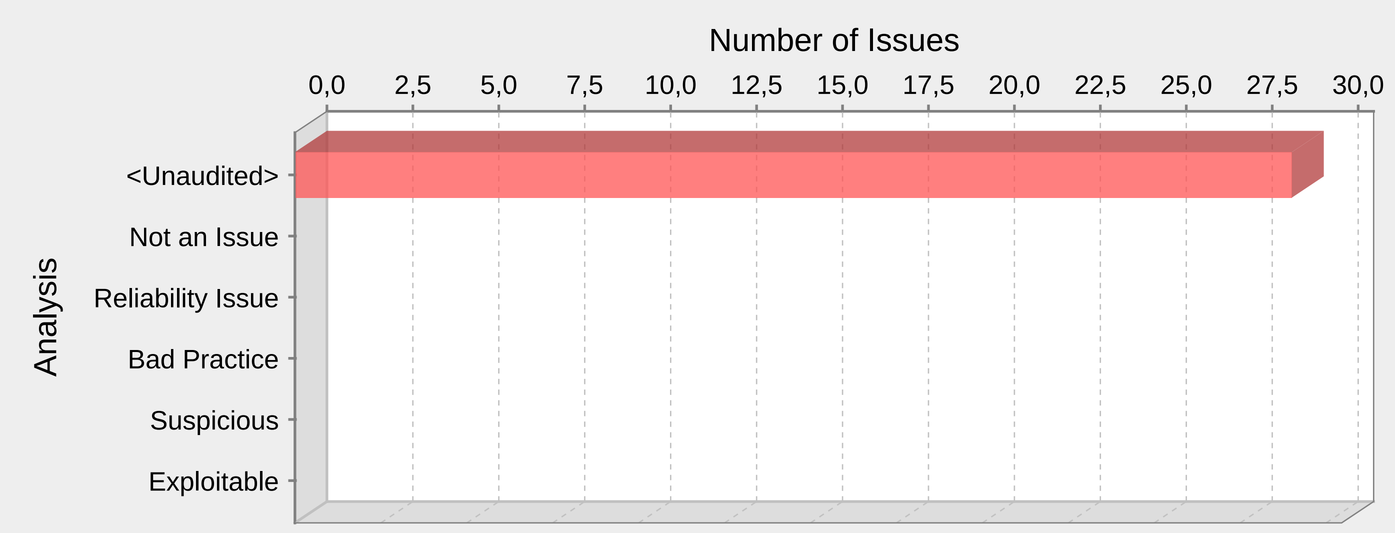
Abstract:	The function tor_vsnprintf() in compat.c might be able to write outside the bounds of allocated memory on line 374, which could corrupt data, cause the program to crash, or lead to the execution of malicious code.		
Source:	util.c:1771 recv()		
Sink:	compat.c:374 Assignment to str[]()		

memarea.c, line 292 (Buffer Overflow)			
Fortify Priority:	Critical	**Folder**	Critical
Kingdom:	Input Validation and Representation		
Abstract:	The function memarea_strndup() in memarea.c might be able to write outside the bounds of allocated memory on line 292, which could corrupt data, cause the program to crash, or lead to the execution of malicious code.		
Source:	util.c:1773 read()		
Sink:	memarea.c:292 memcpy()		

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

memarea.c, line 292 (Buffer Overflow)			
Fortify Priority:	Critical	**Folder**	Critical
Kingdom:	Input Validation and Representation		
Abstract:	The function memarea_strndup() in memarea.c might be able to write outside the bounds of allocated memory on line 292, which could corrupt data, cause the program to crash, or lead to the execution of malicious code.		
Source:	util.c:1771 recv()		
Sink:	memarea.c:292 memcpy()		

Category: Illegal Pointer Value (29 Issues)



Abstract:

This function can return a pointer to memory outside of the buffer to be searched. Subsequent operations on the pointer can have unintended consequences.

Explanation:

This function can return a pointer to memory outside the bounds of the buffer to be searched under either of the following circumstances:

- An attacker can control the contents of the buffer to be searched

- An attacker can control the value for which to search

Example: The following short program uses an untrusted command line argument as the search buffer in a call to memmem().

```
int main(int argc, char** argv) {
char* ret = memmem(argv[0], MAX_PATH, "xx", 2);
printf("%s\n", ret);
}
```

The program is meant to print a substring of argv[0], but it may end up printing some portion of memory above argv[0].

This issue is similar to a buffer overflow resulting from the use of a "safe" string function (such as strncpy()) where the size argument is allowed to get out of sync with the size of the buffer.

Recommendations:

Make sure that the size argument passed to memmem() always matches the size of the buffer to be searched. Do not rely on untrusted data to determine the buffer size. If the buffer you are searching is a null-terminated string, then another option is to make use of the strchr() or strrchr() functions, which properly stop searching when a NULL termination character is encountered.

compat.c, line 488 (Illegal Pointer Value)

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

Abstract:	The call to memmem() on line 488 can return a pointer to memory outside of the buffer to be searched. Subsequent operations on the pointer could have unintended consequences.
Source:	compat.c:200 mmap()
Sink:	compat.c:488 memmem()

compat.c, line 488 (Illegal Pointer Value)

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

Abstract:	The call to memmem() on line 488 can return a pointer to memory outside of the buffer to be searched. Subsequent operations on the pointer could have unintended consequences.
Source:	util.c:1773 read()

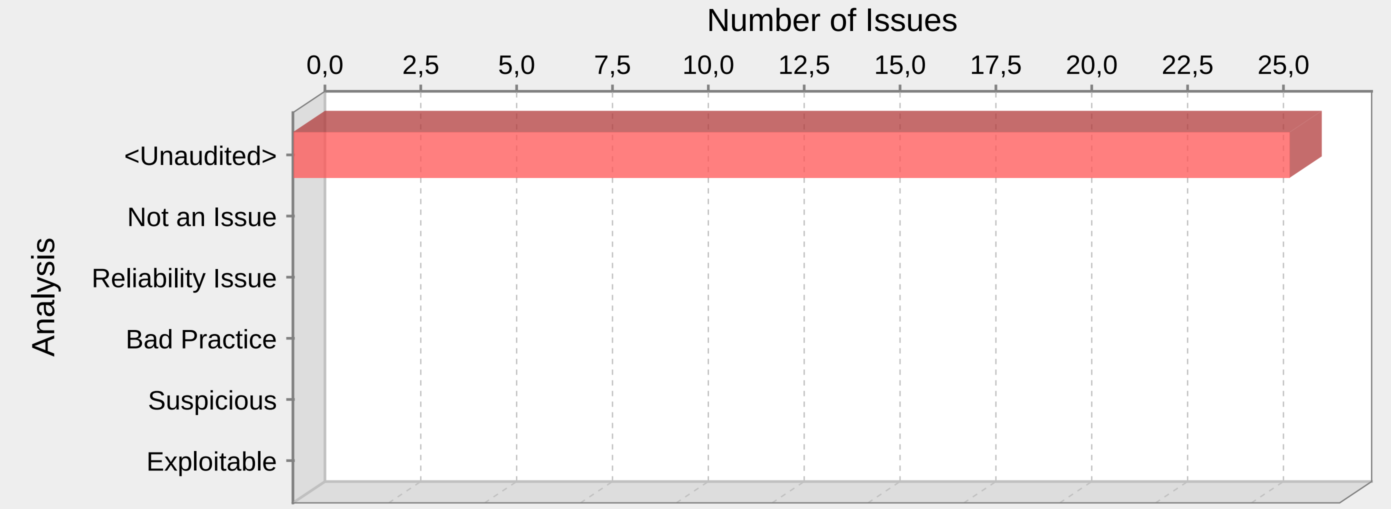
Sink:	compat.c:488 memmem()

routerparse.c, line 1139 (Illegal Pointer Value)			
Fortify Priority:	Low	Folder	Low
Kingdom:	Input Validation and Representation		
Abstract:	The call to memchr() on line 1139 can return a pointer to memory outside of the buffer to be searched. Subsequent operations on the pointer could have unintended consequences.		
Source:	util.c:1773 read()		
Sink:	routerparse.c:1139 memchr()		

routerparse.c, line 1139 (Illegal Pointer Value)			
Fortify Priority:	Low	Folder	Low
Kingdom:	Input Validation and Representation		
Abstract:	The call to memchr() on line 1139 can return a pointer to memory outside of the buffer to be searched. Subsequent operations on the pointer could have unintended consequences.		
Source:	compat.c:200 mmap()		
Sink:	routerparse.c:1139 memchr()		

compat.c, line 488 (Illegal Pointer Value)			
Fortify Priority:	Low	Folder	Low
Kingdom:	Input Validation and Representation		
Abstract:	The call to memmem() on line 488 can return a pointer to memory outside of the buffer to be searched. Subsequent operations on the pointer could have unintended consequences.		
Source:	util.c:1771 recv()		
Sink:	compat.c:488 memmem()		

Category: Poor Style: Value Never Read (26 Issues)



Abstract:

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

Explanation:

This variable's value is not used. After the assignment, 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.

```
r = getName();
r = getNewBuffer(buf);
```

Recommendations:

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

compat.c, line 1642 (Poor Style: Value Never Read)

Fortify Priority:	Low	**Folder**	Low
Kingdom:	Code Quality		
Abstract:	The function tor_disable_debugger_attach() in compat.c never uses the value it assigns to the variable attempted on line 1642.		
Sink:	compat.c:1642 VariableAccess: attempted()		

compat.c, line 1635 (Poor Style: Value Never Read)

Fortify Priority:	Low	**Folder**	Low
Kingdom:	Code Quality		
Abstract:	The function tor_disable_debugger_attach() in compat.c never uses the value it assigns to the variable r on line 1635.		
Sink:	compat.c:1635 VariableAccess: r()		

compat.c, line 1636 (Poor Style: Value Never Read)

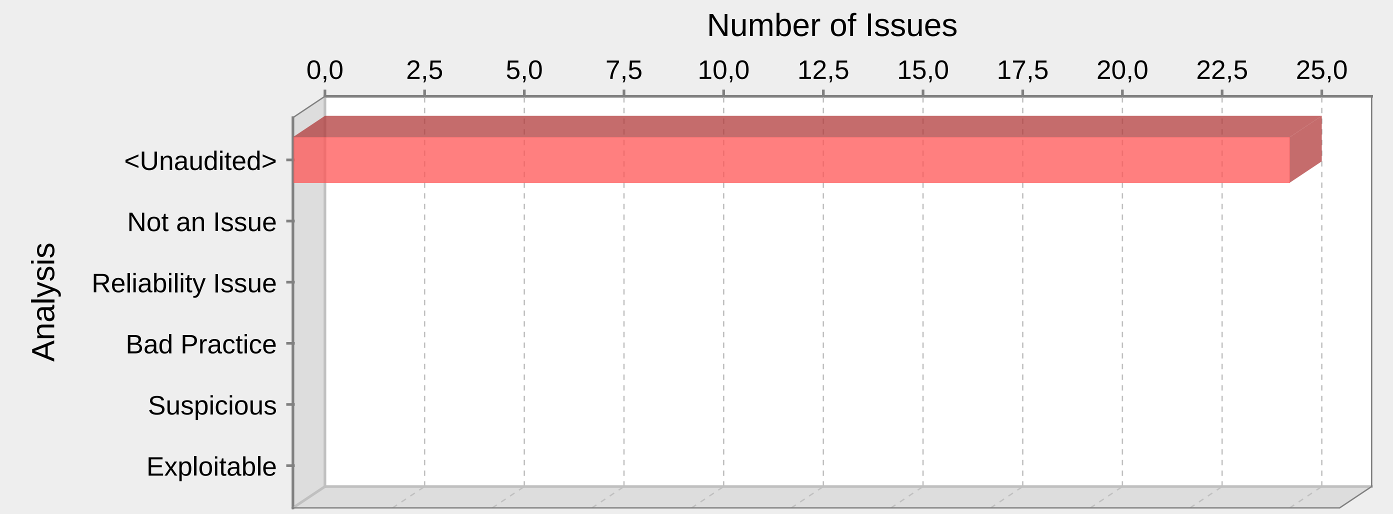
Fortify Priority:	Low	**Folder**	Low
Kingdom:	Code Quality		
Abstract:	The function tor_disable_debugger_attach() in compat.c never uses the value it assigns to the variable attempted on line 1636.		
Sink:	compat.c:1636 VariableAccess: attempted()		

util.c, line 407 (Poor Style: Value Never Read)

Fortify Priority:	Low	**Folder**	Low
Kingdom:	Code Quality		
Abstract:	The function tor_log2() in util.c never uses the value it assigns to the variable u64 on line 407.		
Sink:	util.c:407 VariableAccess: u64()		

container.c, line 522 (Poor Style: Value Never Read)			
Fortify Priority:	Low	**Folder**	Low
Kingdom:	Code Quality		
Abstract:	The function smartlist_get_most_frequent() in container.c never uses the value it assigns to the variable most_frequent_count on line 522.		
Sink:	container.c:522 VariableAccess: most_frequent_count()		

Category: Dangerous Function: strcpy() (25 Issues)



Abstract:

Functions that cannot be used safely should never be used.

Explanation:

Certain functions behave in dangerous ways regardless of how they are used. Functions in this category were often implemented without taking security concerns into account.

The strcpy() function is unsafe because it assumes that its input is null terminated and that there is sufficient memory allocated to accomodate the contents of the source buffer in the destination buffer. In practice, the conditions that must be met to use strcpy() are often too difficult to meet, primarily because they are inherently distinct from the invocation of strcpy.

Recommendations:

Functions that cannot be used safely should never be used. If any of these functions occur in new or legacy code, they must be removed and replaced with safe counterparts.

Replace all calls to strcpy() and similar functions with their bounded counterparts, such as strncpy(). On Windows(R) platforms, consider using functions defined in strsafe.h, such as StringCbCopy(), which takes a buffer size in bytes, or StringCchCopy(), which takes a buffer size in characters. On BSD Unix systems strlcpy() can be used safely because it behaves the same as strncpy() except that it always null terminates its destination buffer. On other systems, always replace instances of strcpy(d, s) with strncpy(d, s, SIZE_D) to check bounds properly and prevent strncpy() from overflowing the destination buffer. For example, if d is a stack-allocated buffer, then SIZE_D can be calculated using sizeof(d).

Tips:

1. Replacing less secure functions like strcpy() with their more secure versions, such as strcpy_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 and null termination errors.

test_pt.c, line 37 (Dangerous Function: strcpy())

Fortify Priority:	Low	**Folder**	Low
Kingdom:	API Abuse		
Abstract:	The function strcpy() cannot be used safely. It should not be used.		
Sink:	test_pt.c:37 strcpy()		

test_pt.c, line 56 (Dangerous Function: strcpy())

Fortify Priority:	Low	**Folder**	Low
Kingdom:	API Abuse		
Abstract:	The function strcpy() cannot be used safely. It should not be used.		
Sink:	test_pt.c:56 strcpy()		

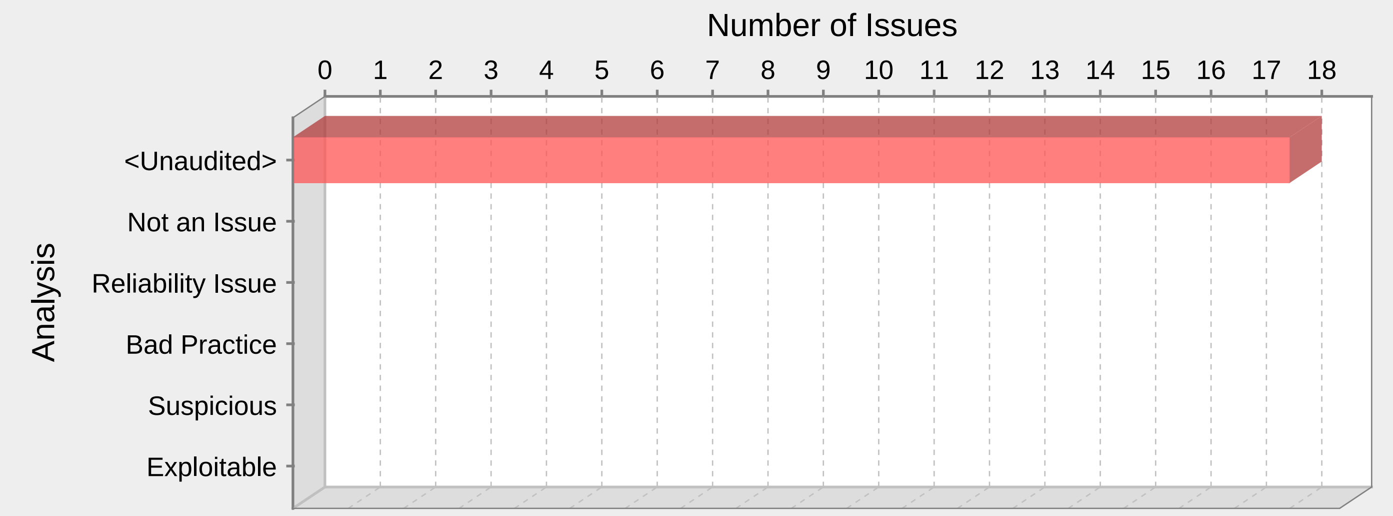
test_pt.c, line 43 (Dangerous Function: strcpy())

Fortify Priority:	Low	**Folder**	Low
Kingdom:	API Abuse		
Abstract:	The function strcpy() cannot be used safely. It should not be used.		
Sink:	test_pt.c:43 strcpy()		

test_pt.c, line 49 (Dangerous Function: strcpy())			
Fortify Priority:	Low	**Folder**	Low
Kingdom:	API Abuse		
Abstract:	The function strcpy() cannot be used safely. It should not be used.		
Sink:	test_pt.c:49 strcpy()		

test_pt.c, line 31 (Dangerous Function: strcpy())			
Fortify Priority:	Low	**Folder**	Low
Kingdom:	API Abuse		
Abstract:	The function strcpy() cannot be used safely. It should not be used.		
Sink:	test_pt.c:31 strcpy()		

Category: Unchecked Return Value (18 Issues)



Abstract:

Ignoring a method's return value can cause the program to overlook unexpected states and conditions.

Explanation:

Just about every serious attack on a software system begins with the violation of a programmer's assumptions. After the attack, the programmer's assumptions seem flimsy and poorly founded, but before an attack many programmers would defend their assumptions well past the end of their lunch break.

Two dubious assumptions that are easy to spot in code are "this function call can never fail" and "it doesn't matter if this function call fails". When a programmer ignores the return value from a function, they implicitly state that they are operating under one of these assumptions.

Example: Consider the following code:

```
char buf[10], cp_buf[10];
fgets(buf, 10, stdin);
strcpy(cp_buf, buf);
```

The programmer expects that when fgets() returns, buf will contain a null-terminated string of length 9 or less. But if an I/O error occurs, fgets() will not null-terminate buf. Furthermore, if the end of the file is reached before any characters are read, fgets() returns without writing anything to buf. In both of these situations, fgets() signals that something unusual has happened by returning NULL, but in this code, the warning will not be noticed. The lack of a null terminator in buf can result in a buffer overflow in the subsequent call to strcpy().

Recommendations:

If a function can return an error code or any other evidence of its success or failure, always check for the error condition, even if there is no obvious way for it to occur. In addition to preventing security errors, many initially mysterious bugs have eventually led back to a failed system call with an ignored return value.

Create an easy to use and standard way for dealing with failure in your application. If error handling is straightforward, programmers will be less inclined to omit it. One approach to standardized error handling is to write wrappers around commonly-used functions that check and handle error conditions without additional programmer intervention. When wrappers are implemented and adopted, the use of non-wrapped equivalents can be prohibited and enforced by using custom rules.

Tips:

1. Watch out for programmers who want to explain away this type of issue by saying "that can never happen because ...". Chances are good that they have developed their intuition about the way the system works by using their development workstation. If your software will eventually run under different operating systems, operating system versions, hardware configurations, or runtime environments, their intuition may not apply.

compat.c, line 186 (Unchecked Return Value)

Fortify Priority:	Low	**Folder**	Low
Kingdom:	API Abuse		
Abstract:	The function tor_mmap_file() in compat.c ignores the value returned by lseek() on line 186, which might cause the program to overlook unexpected states and conditions.		
Sink:	compat.c:186 lseek()		

compat.c, line 221 (Unchecked Return Value)

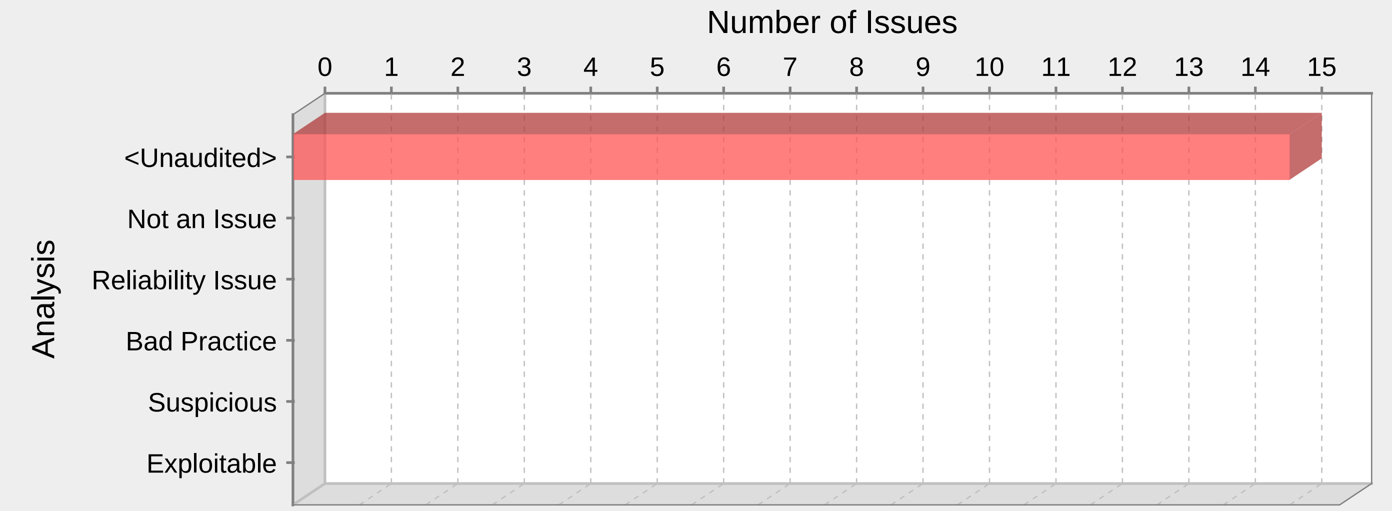
Fortify Priority:	Low	**Folder**	Low
Kingdom:	API Abuse		
Abstract:	The function tor_munmap_file() in compat.c ignores the value returned by munmap() on line 221, which might cause the program to overlook unexpected states and conditions.		
Sink:	compat.c:221 munmap()		

config.c, line 7252 (Unchecked Return Value)			
Fortify Priority:	Low	**Folder**	Low
Kingdom:	API Abuse		
Abstract:	The function remove_file_if_very_old() in config.c ignores the value returned by unlink() on line 7252, which might cause the program to overlook unexpected states and conditions.		
Sink:	config.c:7252 unlink()		

util.c, line 2139 (Unchecked Return Value)			
Fortify Priority:	Low	**Folder**	Low
Kingdom:	API Abuse		
Abstract:	The function finish_writing_to_file_impl() in util.c ignores the value returned by unlink() on line 2139, which might cause the program to overlook unexpected states and conditions.		
Sink:	util.c:2139 unlink()		

config.c, line 6903 (Unchecked Return Value)			
Fortify Priority:	Low	**Folder**	Low
Kingdom:	API Abuse		
Abstract:	The function or_state_save_broken() in config.c ignores the value returned by unlink() on line 6903, which might cause the program to overlook unexpected states and conditions.		
Sink:	config.c:6903 unlink()		

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.

config.c, line 6975 (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://or/config.c###6887###6887###0###0">or_state_save_broken()</a> can be exploited to launch a privilege escalation attack.		
Sink:	config.c:6975 or_state_save_broken(fname) : Symbolic filename fname used to operate on a file()		

config.c, line 6448 (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:///usr/include/stdio.h###157###157###0###0">rename()</a> can be exploited to launch a privilege escalation attack.		
Sink:	config.c:6448 rename(fname, ...) : Symbolic filename fname used to operate on a file()		

config.c, line 6989 (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://or/config.c###6887###6887###0###0">or_state_save_broken()</a> can be exploited to launch a privilege escalation attack.		
Sink:	config.c:6989 or_state_save_broken(fname) : Symbolic filename fname used to operate on a file()		

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

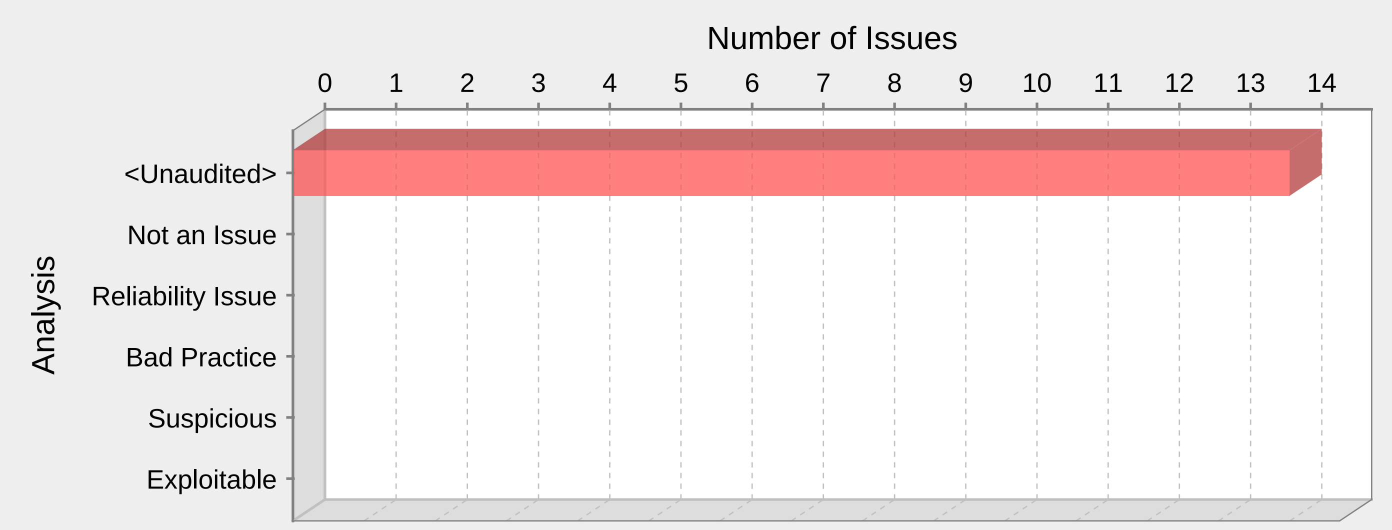
Fortify Priority:	High	**Folder**	High
Kingdom:	Time and State		

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()

config.c, line 7252 (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:///usr/include/sys/stat.h###219###219###0###0">stat()</a> and <a href="location:///usr/include/unistd.h###842###842###0###0">unlink()</a> can be exploited to launch a privilege escalation attack.		
Sink:	config.c:7252 unlink(fname) : Symbolic filename fname used to operate on a file()		

Category: Integer Overflow (14 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:	Low	**Folder**	Low
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:	eventdns.c:2787 gethostname()		
Sink:	util.c:144 malloc()		

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()		
Sink:	util.c:144 malloc()		

util.c, line 144 (Integer Overflow)

Fortify Priority:	Low	**Folder**	Low
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:	util.c:2901 readdir()		
Sink:	util.c:144 malloc()		

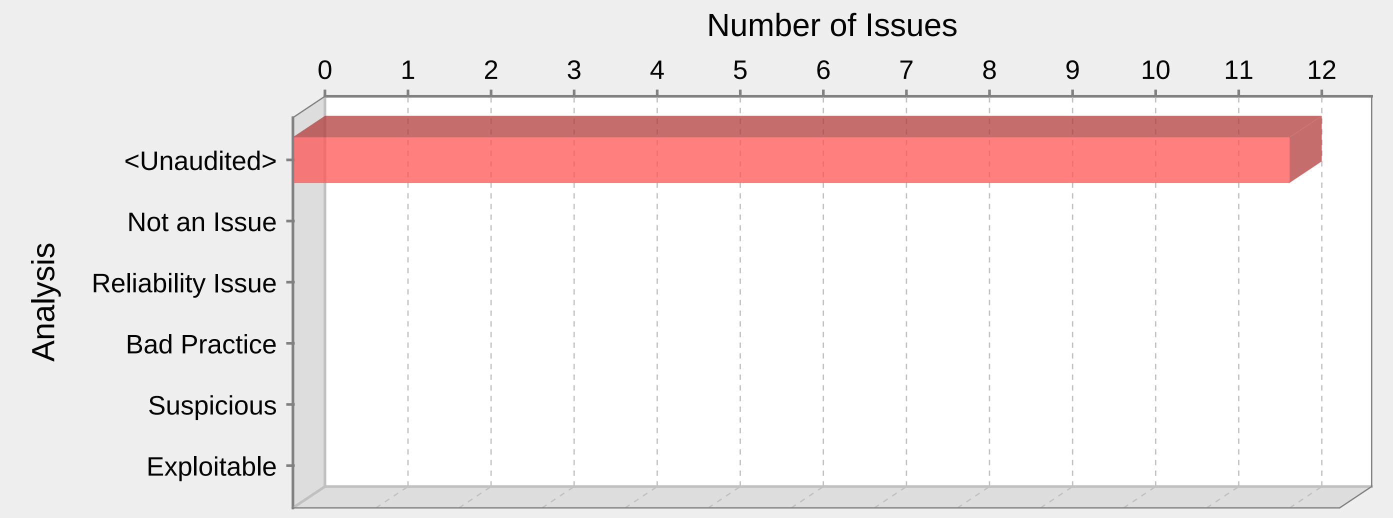
util.c, line 144 (Integer Overflow)

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:	eventdns.c:2787 gethostname()		

Sink:	util.c:144 malloc()

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:	util.c:1773 read()		
Sink:	util.c:144 malloc()		

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.

connection_edge.c, line 3180 (Null Dereference)

Fortify Priority:	High	**Folder**	High
Kingdom:	Code Quality		
Abstract:	The function connection_exit_begin_conn() in connection_edge.c can crash the program by dereferencing a null pointer on line 3180.		
Sink:	connection_edge.c:3180 Dereferenced : or_circ()		

control.c, line 2500 (Null Dereference)

Fortify Priority:	High	**Folder**	High
Kingdom:	Code Quality		
Abstract:	The function handle_control_extendcircuit() in control.c can crash the program by dereferencing a null pointer on line 2500.		
Sink:	control.c:2500 Dereferenced : circ()		

circuitlist.c, line 1490 (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 1490.
Sink:	circuitlist.c:1490 Dereferenced : or_circ()

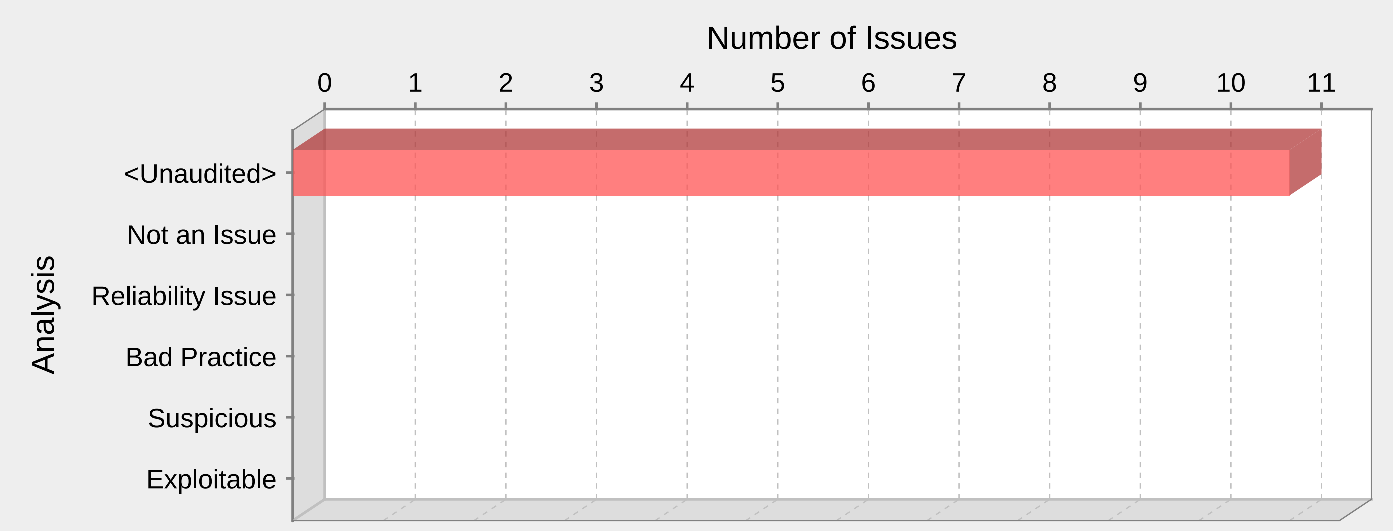
control.c, line 2495 (Null Dereference)

Fortify Priority:	High	**Folder**	High
Kingdom:	Code Quality		
Abstract:	The function handle_control_extendcircuit() in control.c can crash the program by dereferencing a null pointer on line 2495.		
Sink:	control.c:2495 Dereferenced : circ()		

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()		

Category: Heap Inspection (11 Issues)



Abstract:

Do not use realloc() to resize buffers that store sensitive information. The function might leave a copy of the sensitive information stranded in memory where it cannot be overwritten.

Explanation:

Heap inspection vulnerabilities occur when sensitive data, such as a password or an encryption key, can be exposed to an attacker because they are not removed from memory.

The realloc() function is commonly used to increase the size of a block of allocated memory. This operation often requires copying the contents of the old memory block into a new and larger block. This operation leaves the contents of the original block intact but inaccessible to the program, preventing the program from being able to scrub sensitive data from memory. If an attacker can later examine the contents of a memory dump, the sensitive data could be exposed.

Example: The following code calls realloc() on a buffer containing sensitive data:

```
cleartext_buffer = get_secret();
...
cleartext_buffer = realloc(cleartext_buffer, 1024);
...
scrub_memory(cleartext_buffer, 1024);
```

There is an attempt to scrub the sensitive data from memory, but realloc() is used, so a copy of the data can still be exposed in the memory originally allocated for cleartext_buffer.

Recommendations:

If sensitive information is stored in memory and there is a risk of an attacker having access to a memory dump, replace realloc() with explicit calls to malloc(), memcpy(), and free(). This approach gives you the opportunity to safely scrub the contents of the original buffer before it is freed.

Tips:

1. Depending upon your system, secure memory might not be a concern. If you are satisfied that an attacker will never be able to inspect the heap or if no sensitive data is manipulated by the program, you can use AuditGuide to filter out this category.

util.c, line 218 (Heap Inspection)

Fortify Priority:	Low	**Folder**	Low
Kingdom:	API Abuse		
Abstract:	Do not use realloc() to resize buffers that store sensitive information. The function might leave a copy of the sensitive information stranded in memory where it cannot be overwritten.		
Sink:	util.c:218 realloc()		

container.c, line 956 (Heap Inspection)

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

Abstract:	Do not use realloc() to resize buffers that store sensitive information. The function might leave a copy of the sensitive information stranded in memory where it cannot be overwritten.		
Sink:	container.c:956 realloc()		

circuitlist.c, line 78 (Heap Inspection)

Fortify Priority:	Low	**Folder**	Low
Kingdom:	API Abuse		
Abstract:	Do not use realloc() to resize buffers that store sensitive information. The function might leave a copy of the sensitive information stranded in memory where it cannot be overwritten.		
Sink:	circuitlist.c:78 realloc()		

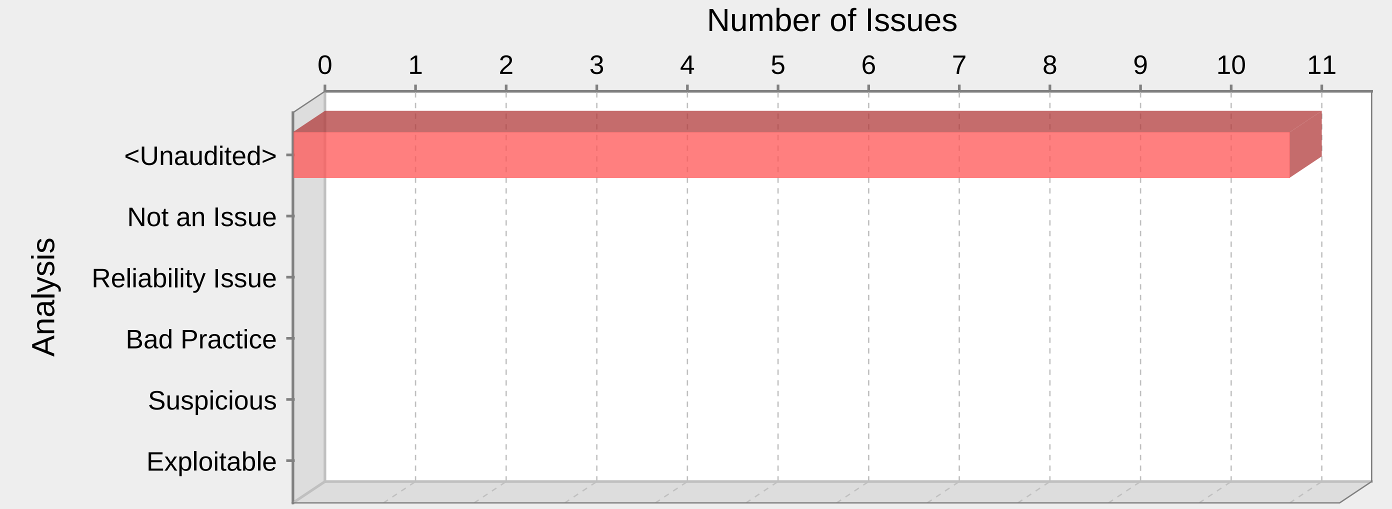
container.c, line 961 (Heap Inspection)

Fortify Priority:	Low	**Folder**	Low
Kingdom:	API Abuse		
Abstract:	Do not use realloc() to resize buffers that store sensitive information. The function might leave a copy of the sensitive information stranded in memory where it cannot be overwritten.		
Sink:	container.c:961 realloc()		

dns.c, line 203 (Heap Inspection)

Fortify Priority:	Low	**Folder**	Low
Kingdom:	API Abuse		
Abstract:	Do not use realloc() to resize buffers that store sensitive information. The function might leave a copy of the sensitive information stranded in memory where it cannot be overwritten.		
Sink:	dns.c:203 realloc()		

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.

dirserv.c, line 3138 (Key Management: Hardcoded Encryption Key)

Fortify Priority:	Critical	**Folder**	Critical
Kingdom:	Security Features		
Abstract:	Encryption keys can compromise system security in a way that cannot be easily remedied.		
Sink:	dirserv.c:3138 FunctionCall: strcmp()		

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

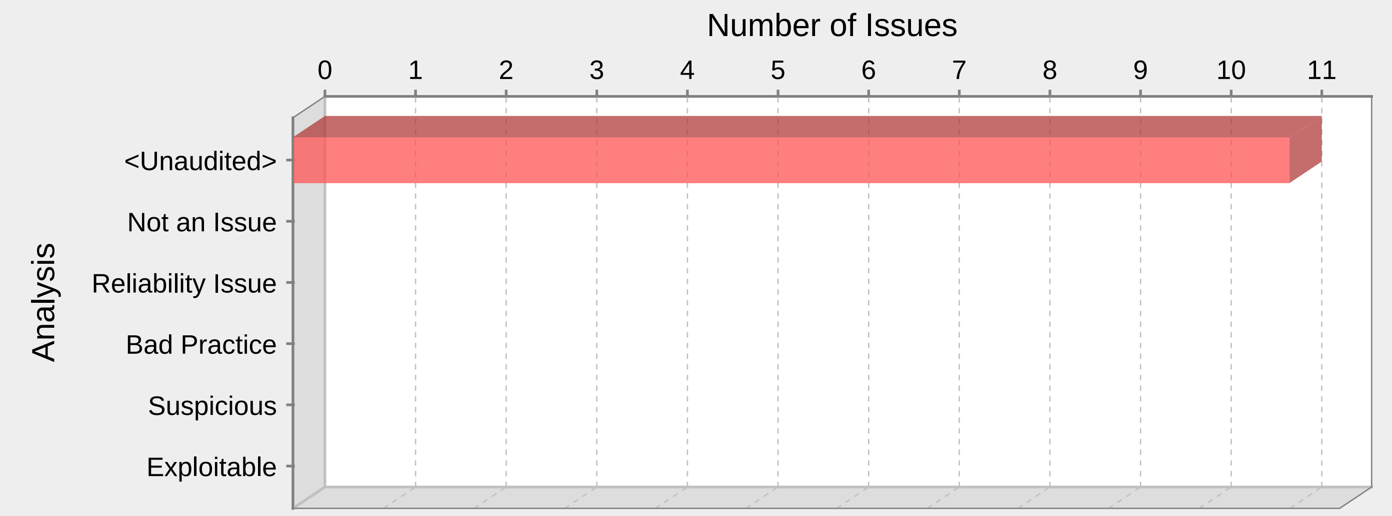
Fortify Priority:	Critical	**Folder**	Critical
Kingdom:	Security Features		
Abstract:	Encryption keys can compromise system security in a way that cannot be easily remedied.		
Sink:	config.c:6823 FunctionCall: strcmp()		

dirserv.c, line 3057 (Key Management: Hardcoded Encryption Key)

Fortify Priority:	Critical	**Folder**	Critical
Kingdom:	Security Features		
Abstract:	Encryption keys can compromise system security in a way that cannot be easily remedied.		

Sink:	dirserv.c:3057 FunctionCall: strcmp()		
dirserv.c, line 3064 (Key Management: Hardcoded Encryption Key)			
Fortify Priority:	Critical	**Folder**	Critical
Kingdom:	Security Features		
Abstract:	Encryption keys can compromise system security in a way that cannot be easily remedied.		
Sink:	dirserv.c:3064 FunctionCall: strcmp()		
config.c, line 7098 (Key Management: Hardcoded Encryption Key)			
Fortify Priority:	Critical	**Folder**	Critical
Kingdom:	Security Features		
Abstract:	Encryption keys can compromise system security in a way that cannot be easily remedied.		
Sink:	config.c:7098 FunctionCall: strcmp()		

Category: Redundant Null Check (11 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. Specifically, dereference-after-check errors occur when a program makes an explicit check for null, but proceeds to dereference the pointer when it is known to be null. Errors of this type are often the result of a typo or programmer oversight.

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 mount a denial of service attack or to cause the application to reveal debugging information that will be valuable in planning subsequent attacks.

Example 1: In the following code, the programmer confirms that the variable ptr is NULL and subsequently dereferences it erroneously. If ptr is NULL when it is checked in the if statement, then a null dereference will occur, thereby causing a segmentation fault.

```
if (ptr == null) {
ptr->field = val;
...
}
```

Example 2: In the following code, the programmer forgets that the string '\0' actually 0 or NULL, thereby dereferencing a null pointer and causing a segmentation fault.

```
if (ptr == '\0') {
*ptr = val;
...
}
```

Recommendations:

Security problems caused by dereferencing NULL pointers almost always take the form of denial of service attacks. If an attacker can consistently trigger a null pointer dereference then other users may be prevented from gaining legitimate access to the application. Apart from situations where an attacker can deliberately trigger a segmentation fault, dereferencing a null point may cause sporadic crashes that can be hard to track down.

connection_edge.c, line 1004 (Redundant Null Check)

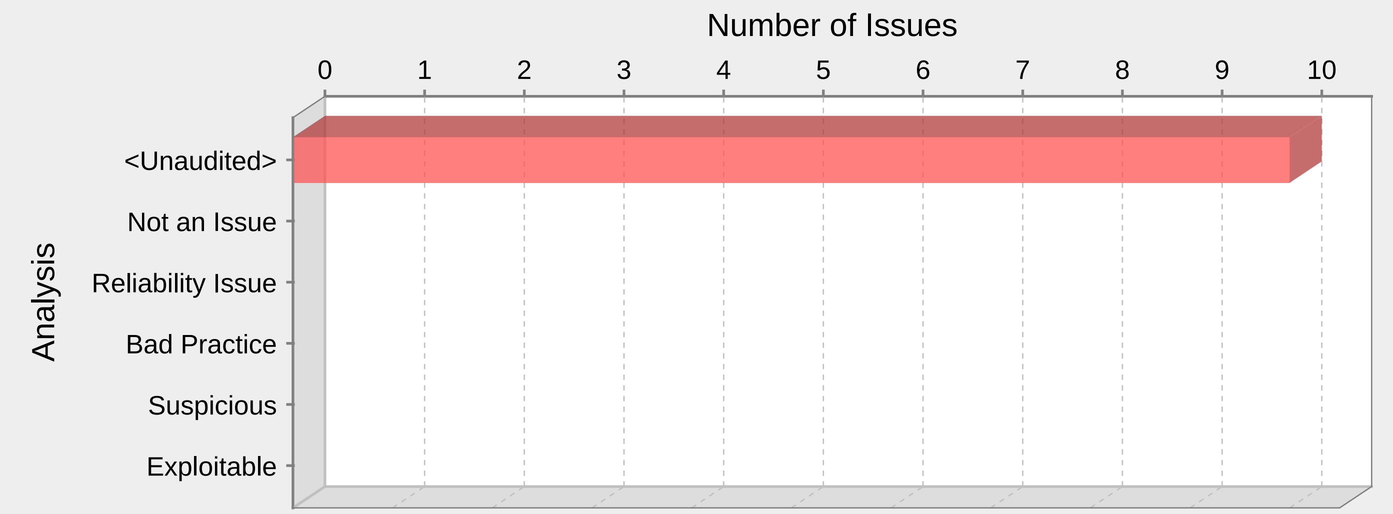
Fortify Priority:	Low	**Folder**	Low
Kingdom:	Code Quality		
Abstract:	The function addressmap_clear_invalid_automaps() in connection_edge.c can crash the program by dereferencing a null pointer on line 1004.		
Sink:	connection_edge.c:1004 Dereferenced : suffixes()		

compat.c, line 2397 (Redundant Null Check)

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

Abstract:	The function correct_tm() in compat.c can crash the program by dereferencing a null pointer on line 2397.		
Sink:	compat.c:2397 Dereferenced : r()		
rephist.c, line 1691 (Redundant Null Check)			
Fortify Priority:	Low	**Folder**	Low
Kingdom:	Code Quality		
Abstract:	The function rep_hist_load_bwhist_state_section() in rephist.c can crash the program by dereferencing a null pointer on line 1691.		
Sink:	rephist.c:1691 Dereferenced : s_maxima()		
policies.c, line 519 (Redundant Null Check)			
Fortify Priority:	Low	**Folder**	Low
Kingdom:	Code Quality		
Abstract:	The function cmp_addr_policies() in policies.c can crash the program by dereferencing a null pointer on line 519.		
Sink:	policies.c:519 Dereferenced : b()		
policies.c, line 519 (Redundant Null Check)			
Fortify Priority:	Low	**Folder**	Low
Kingdom:	Code Quality		
Abstract:	The function cmp_addr_policies() in policies.c can crash the program by dereferencing a null pointer on line 519.		
Sink:	policies.c:519 Dereferenced : a()		

Category: Missing Check against Null (10 Issues)



Abstract:

The program can dereference a null pointer because it does not check the return value of a function that might return null.

Explanation:

Just about every serious attack on a software system begins with the violation of a programmer's assumptions. After the attack, the programmer's assumptions seem flimsy and poorly founded, but before an attack many programmers would defend their assumptions well past the end of their lunch break.

Two dubious assumptions that are easy to spot in code are "this function call can never fail" and "it doesn't matter if this function call fails". When a programmer ignores the return value from a function, they implicitly state that they are operating under one of these assumptions.

Example 1: The following code does not check to see if memory allocation succeeded before attempting to use the pointer returned by malloc().

```
buf = (char*) malloc(req_size);
strncpy(buf, xfer, req_size);
```

The traditional defense of this coding error is:

"If my program runs out of memory, it will fail. It doesn't matter whether I handle the error or simply allow the program to die with a segmentation fault when it tries to dereference the null pointer."

This argument ignores three important considerations:

- Depending upon the type and size of the application, it may be possible to free memory that is being used elsewhere so that execution can continue.

- It is impossible for the program to perform a graceful exit if required. If the program is performing an atomic operation, it can leave the system in an inconsistent state.

- The programmer has lost the opportunity to record diagnostic information. Did the call to malloc() fail because req_size was too large or because there were too many requests being handled at the same time? Or was it caused by a memory leak that has built up over time? Without handling the error, there is no way to know.

Recommendations:

If a function can return an error code or any other evidence of its success or failure, always check for the error condition, even if there is no obvious way for it to occur. In addition to preventing security errors, many initially mysterious bugs have eventually led back to a failed system call with an ignored return value.

Create an easy to use and standard way for dealing with failure in your application. If error handling is straightforward, programmers will be less inclined to omit it. One approach to standardized error handling is to write wrappers around commonly-used functions that check and handle error conditions without additional programmer intervention. When wrappers are implemented and adopted, the use of non-wrapped equivalents can be prohibited and enforced by using custom rules.

Example 2: The following code implements a wrapper around malloc() that checks the return value of malloc() against NULL and exits if the memory allocation failed.

```
void *checked_malloc (size_t size) {
void *ptr;
ptr= malloc(size);
if (ptr == NULL) {
fprintf (stderr, "Out of memory: %s:%d",
__FILE__,__LINE__);
exit(-1);
}
return ptr;
}
```

The example above uses the simplest error handling mechanism available in low memory conditions: it kills the application. Based on the context in which the failure occurs, other behavior may be appropriate. For example, consider whether the application is using memory elsewhere for less important operations that could be freed to make the current allocation succeed. Depending on the application and the system it runs on, waiting for more memory to become available might also be a practical option. Regardless, in most cases the most important piece of error handling to execute in low memory situations are log entries so that the problem can be accurately diagnosed and possibly prevented in the future.

Tips:

1. Watch out for programmers who want to explain away this type of issue by saying "that can never happen because ...". Chances are good that they have developed their intuition about the way the system works by using their development workstation. If your software will eventually run under different operating systems, operating system versions, hardware configurations, or runtime environments, their intuition may not apply.

test_util.c, line 972 (Missing Check against Null)

Fortify Priority:	Low	**Folder**	Low
Kingdom:	API Abuse		
Abstract:	The function test_util_strmisc() in test_util.c can dereference a null pointer on line 972 because it does not check the return value of tor_memmem(), which might return null.		
Sink:	test_util.c:972 val1_ = tor_memmem(...) : tor_memmem may return NULL()		

eventdns.c, line 3004 (Missing Check against Null)

Fortify Priority:	Low	**Folder**	Low
Kingdom:	API Abuse		
Abstract:	The function resolv_conf_parse_line() in eventdns.c can dereference a null pointer on line 3005 because it does not check the return value of strtok_r(), which might return null.		
Sink:	eventdns.c:3004 nameserver = strtok_r(...) : strtok_r may return NULL()		

control.c, line 1541 (Missing Check against Null)

Fortify Priority:	Low	**Folder**	Low
Kingdom:	API Abuse		
Abstract:	The function munge_extrainfo_into_routerinfo() in control.c can dereference a null pointer on line 1542 because it does not check the return value of memchr(), which might return null.		
Sink:	control.c:1541 eol = memchr(...) : memchr may return NULL()		

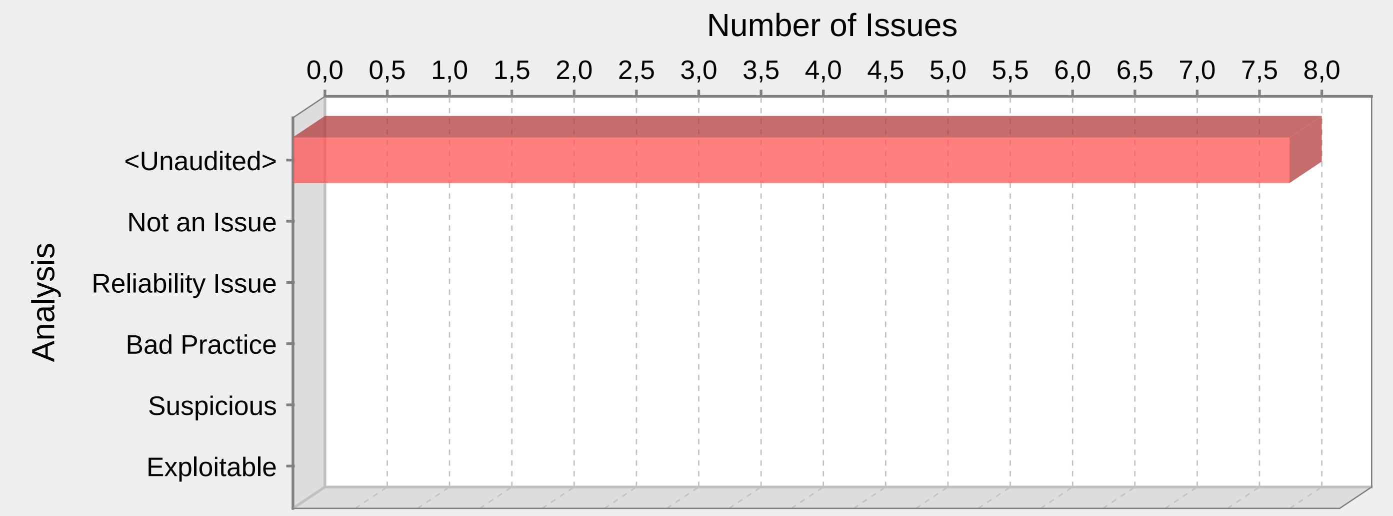
test_util.c, line 973 (Missing Check against Null)

Fortify Priority:	Low	**Folder**	Low
Kingdom:	API Abuse		
Abstract:	The function test_util_strmisc() in test_util.c can dereference a null pointer on line 973 because it does not check the return value of tor_memmem(), which might return null.		
Sink:	test_util.c:973 val1_ = tor_memmem(...) : tor_memmem may return NULL()		

log.c, line 198 (Missing Check against Null)

Fortify Priority:	Low	**Folder**	Low
Kingdom:	API Abuse		
Abstract:	The function _log_prefix() in log.c can dereference a null pointer on line 199 because it does not check the return value of strftime(), which might return null.		
Sink:	log.c:198 n = strftime(...) : strftime may return NULL()		

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 1569 (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.		
Sink:	compat.c:1569 setgid()		

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()		
Sink:	compat.c:1524 setegid()		

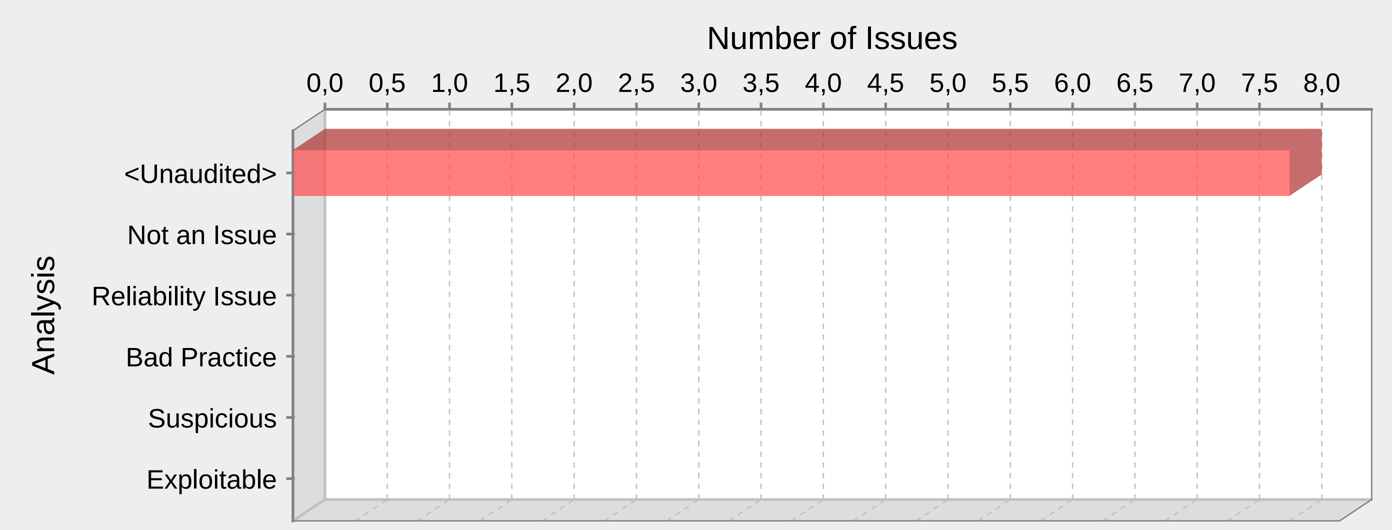
compat.c, line 1542 (Often Misused: Privilege Management)

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()		
Sink:	compat.c:1542 seteuid()		

compat.c, line 1536 (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()		
Sink:	compat.c:1536 setuid()		

compat.c, line 1530 (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()		
Sink:	compat.c:1530 setgid()		

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 138 (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 138, which allows them to access or modify otherwise protected files.		
Source:	tor-checkkey.c:18 main(1)		
Sink:	compat.c:138 open()		

compat.c, line 138 (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 138, which allows them to access or modify otherwise protected files.		
Source:	tor-gencert.c:512 main(1)		
Sink:	compat.c:138 open()		

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-gencert.c:512 main(1)		
Sink:	compat.c:128 open()		

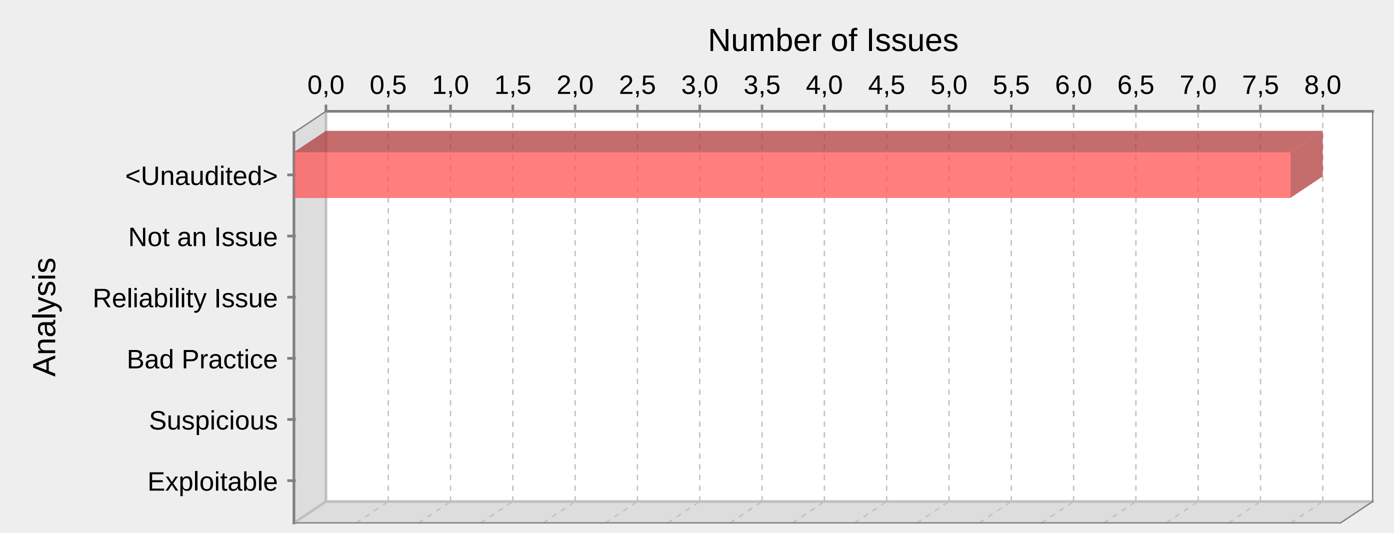
compat.c, line 128 (Path Manipulation)

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)		
Sink:	compat.c:128 open()		

compat.c, line 723 (Path Manipulation)

Fortify Priority:	Critical	**Folder**	Critical
Kingdom:	Input Validation and Representation		
Abstract:	Attackers can control the filesystem path argument to rename() at compat.c line 723, which allows them to access or modify otherwise protected files.		
Source:	tor-gencert.c:512 main(1)		
Sink:	compat.c:723 rename()		

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;
}
```

connection.c, line 986 (Unreleased Resource)

Fortify Priority:	High	**Folder**	High
Kingdom:	Code Quality		
Abstract:	The function connection_listener_new() in connection.c sometimes fails to release a system resource allocated by <a href="location://common/compat.c###1009###1009###0###0">tor_open_socket()</a> on line 986.		
Sink:	connection.c:986 s = tor_open_socket(...)		

util.c, line 2288 (Unreleased Resource)

Fortify Priority:	High	**Folder**	High
Kingdom:	Code Quality		
Abstract:	The function read_file_to_str() in util.c sometimes fails to release a system resource allocated by <a href="location://common/compat.c###124###124###0###0">tor_open_cloexec()</a> on line 2288.		
Sink:	util.c:2288 fd = tor_open_cloexec(...)		

log.c, line 833 (Unreleased Resource)

Fortify Priority:	High	**Folder**	High
Kingdom:	Code Quality		
Abstract:	The function add_file_log() in log.c sometimes fails to release a system resource allocated by <a href="location://common/compat.c###124###124###0###0">tor_open_cloexec()</a> on line 833.		
Sink:	log.c:833 fd = tor_open_cloexec(...)		

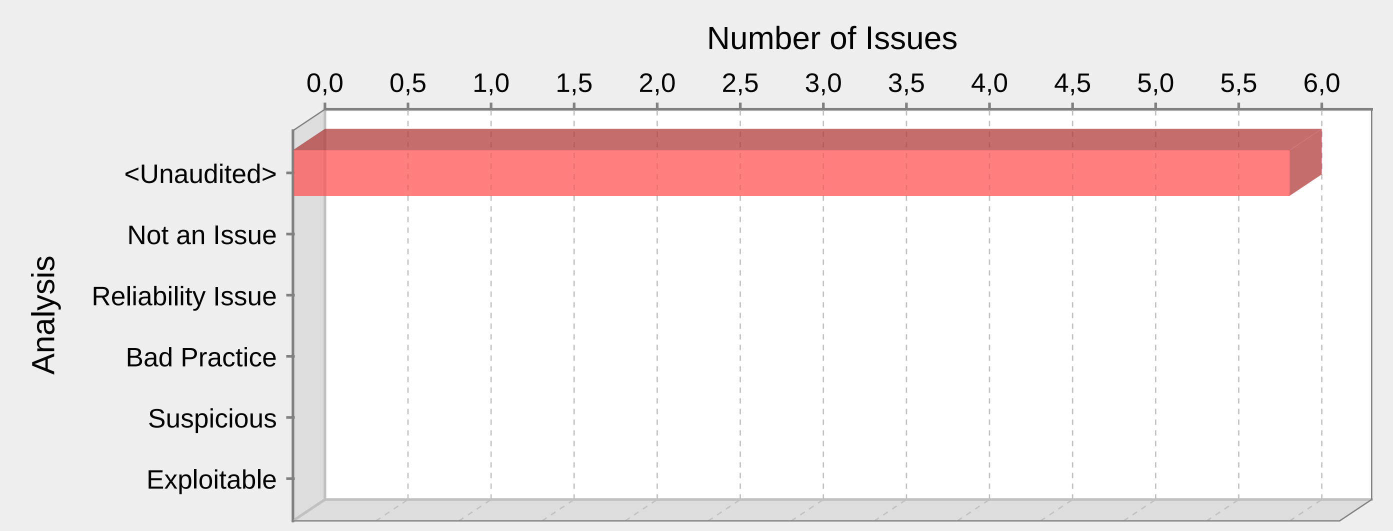
compat.c, line 2878 (Unreleased Resource)

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()

util.c, line 3023 (Unreleased Resource)			
Fortify Priority:	High	**Folder**	High
Kingdom:	Code Quality		
Abstract:	The function finish_daemon() in util.c sometimes fails to release a system resource allocated by <a href="location://common/compat.c###124###124###0###0">tor_open_cloexec()</a> on line 3023.		
Sink:	util.c:3023 nullfd = tor_open_cloexec(...)		

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>()		

main.c, line 2248 (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:2248 sigaction(1, (&action), ...) : Handler (&action) registered for signal <inline expression>()		

main.c, line 2247 (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:2247 sigaction(12, (&action), ...) : Handler (&action) registered for signal <inline expression>()		

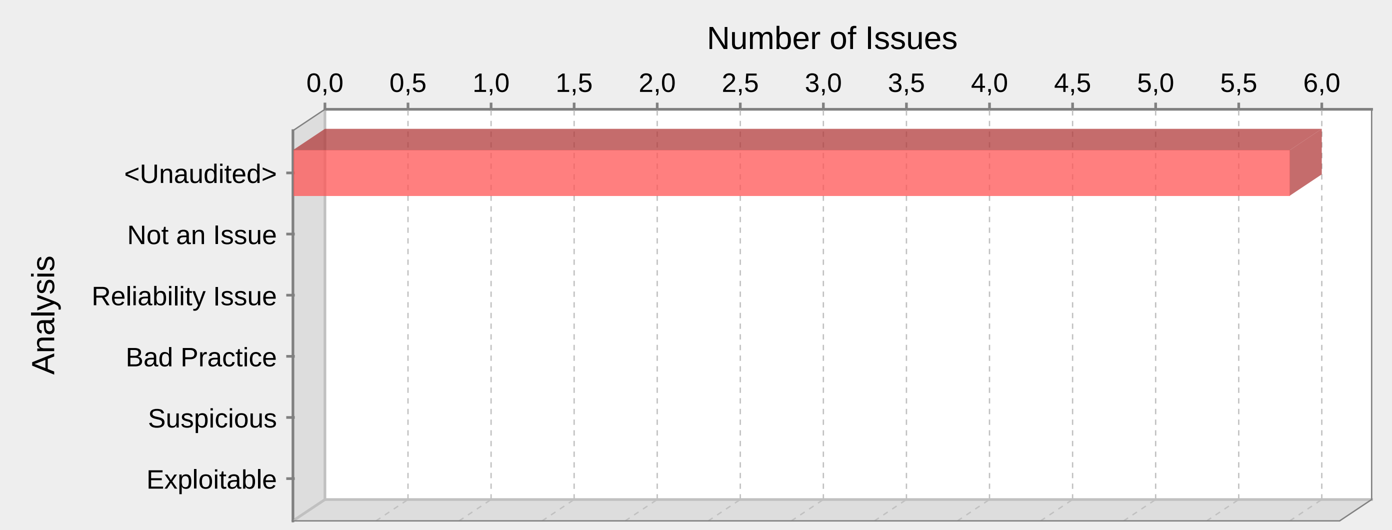
main.c, line 2246 (Race Condition: Signal Handling)

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:2246 sigaction(10, (&action), ...) : Handler (&action) registered for signal <inline expression>()		

main.c, line 2245 (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:2245 sigaction(13, (&action), ...) : Handler (&action) registered for signal <inline expression>()		

Category: Weak Encryption: Inadequate RSA Padding (6 Issues)



Abstract:

The RSA algorithm is used without OAEP padding, thereby making the encryption weak.

Explanation:

When used in practice, RSA is generally combined with some padding scheme. The goal of the padding scheme is to prevent a number of attacks that potentially work against RSA without padding.

Example 1: The following code uses RSA encryption algorithm without appropriate padding.

```
void encrypt_with_rsa(BIGNUM *out, BIGNUM *in, RSA *key) {
u_char *inbuf, *outbuf;
int ilen;
...
ilen = BN_num_bytes(in);
inbuf = xmalloc(ilen);
BN_bn2bin(in, inbuf);
if ((len = RSA_public_encrypt(ilen, inbuf, outbuf, key, RSA_NO_PADDING)) <= 0) {
fatal("encrypt_with_rsa() failed");
}
...
}
```

This category was derived from the Cigital Java Rulepack. http://www.cigital.com/securitypack/

Recommendations:

In order to use RSA securely, the OAEP padding mode (Optimal Asymmetric Encryption Padding) must be used.

Example 2: The following code uses RSA encryption algorithm with OAEP padding.

```
void encrypt_with_rsa(BIGNUM *out, BIGNUM *in, RSA *key) {
u_char *inbuf, *outbuf;
int ilen;
...
ilen = BN_num_bytes(in);
inbuf = xmalloc(ilen);
BN_bn2bin(in, inbuf);
if ((len = RSA_public_encrypt(ilen, inbuf, outbuf, key, RSA_PKCS1_OAEP_PADDING)) <= 0) {
fatal("encrypt_with_rsa() failed");
}
...
}
```

crypto.c, line 957 (Weak Encryption: Inadequate RSA Padding)

Fortify Priority:	Low	**Folder**	Low
Kingdom:	Security Features		
Abstract:	The method crypto_pk_private_sign() in crypto.c uses the RSA algorithm without OAEP padding, thereby making the encryption weak.		
Sink:	crypto.c:957 RSA_private_encrypt()		

crypto.c, line 851 (Weak Encryption: Inadequate RSA Padding)

Fortify Priority:	Low	**Folder**	Low
Kingdom:	Security Features		
Abstract:	The method crypto_pk_private_decrypt() in crypto.c uses the RSA algorithm without OAEP padding, thereby making the encryption weak.		
Sink:	crypto.c:851 RSA_private_decrypt()		

tor-gencert.c, line 471 (Weak Encryption: Inadequate RSA Padding)

Fortify Priority:	Low	**Folder**	Low
Kingdom:	Security Features		
Abstract:	The method generate_certificate() in tor-gencert.c uses the RSA algorithm without OAEP padding, thereby making the encryption weak.		
Sink:	tor-gencert.c:471 RSA_private_encrypt()		

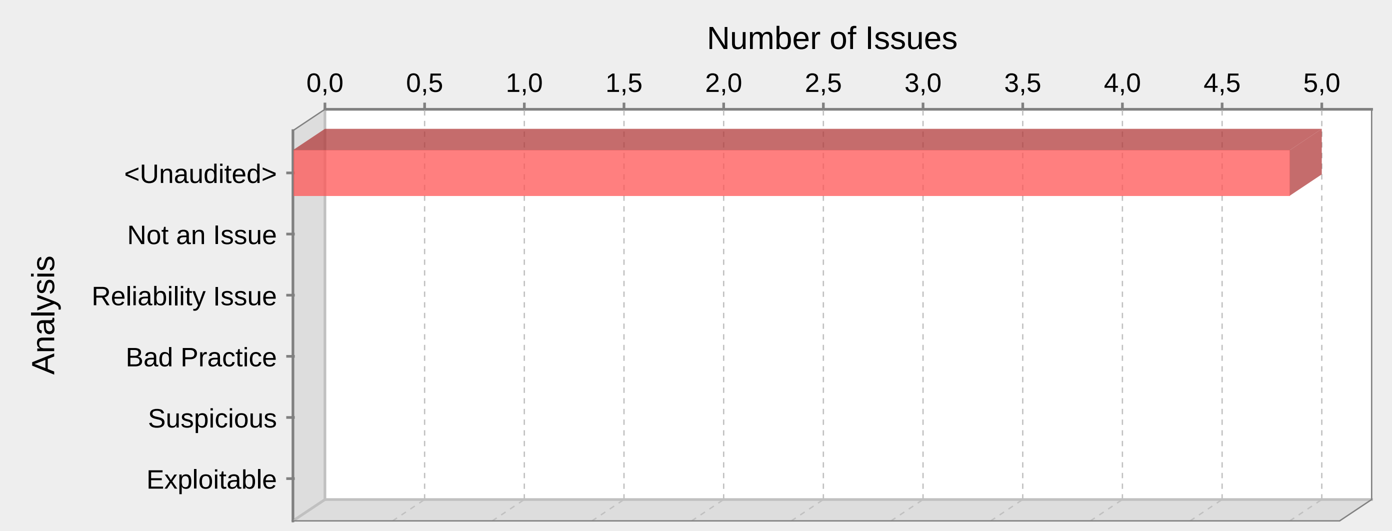
crypto.c, line 882 (Weak Encryption: Inadequate RSA Padding)

Fortify Priority:	Low	**Folder**	Low
Kingdom:	Security Features		
Abstract:	The method crypto_pk_public_checksig() in crypto.c uses the RSA algorithm without OAEP padding, thereby making the encryption weak.		
Sink:	crypto.c:882 RSA_public_decrypt()		

crypto.c, line 816 (Weak Encryption: Inadequate RSA Padding)

Fortify Priority:	Low	**Folder**	Low
Kingdom:	Security Features		
Abstract:	The method crypto_pk_public_encrypt() in crypto.c uses the RSA algorithm without OAEP padding, thereby making the encryption weak.		
Sink:	crypto.c:816 RSA_public_encrypt()		

Category: Password Management: Null Password (5 Issues)



Abstract:

Null passwords can compromise security.

Explanation:

Assigning null to password variables is a bad idea because it can allow attackers to bypass password verification or might indicate that resources are protected by an empty password.

Example: The code below initializes a password variable to null, attempts to read a stored value for the password, and compares it against a user-supplied value.

```
...
char *stored_password = NULL;
readPassword(stored_password);
if(safe_strcmp(stored_password, user_password))
// Access protected resources
...
}
...
```

If readPassword() fails to retrieve the stored password due to a database error or another problem, then an attacker could trivially bypass the password check by providing a null value for user_password.

Recommendations:

Always read stored password values from encrypted, external resources and assign password variables meaningful values. Ensure that sensitive resources are never protected with empty or null passwords.

Starting with Microsoft(R) Windows(R) 2000, Microsoft(R) provides Windows Data Protection Application Programming Interface (DPAPI), which is an OS-level service that protects sensitive application data, such as passwords and private keys [1].

Tips:

1. When identifying null, empty, or hardcoded passwords, default rules only consider fields and variables that contain the word password. However, the Custom Rules Editor provides the Password Management wizard that makes it easy to create rules for detecting password management issues on custom-named fields and variables.

buffers.c, line 1554 (Password Management: Null Password)

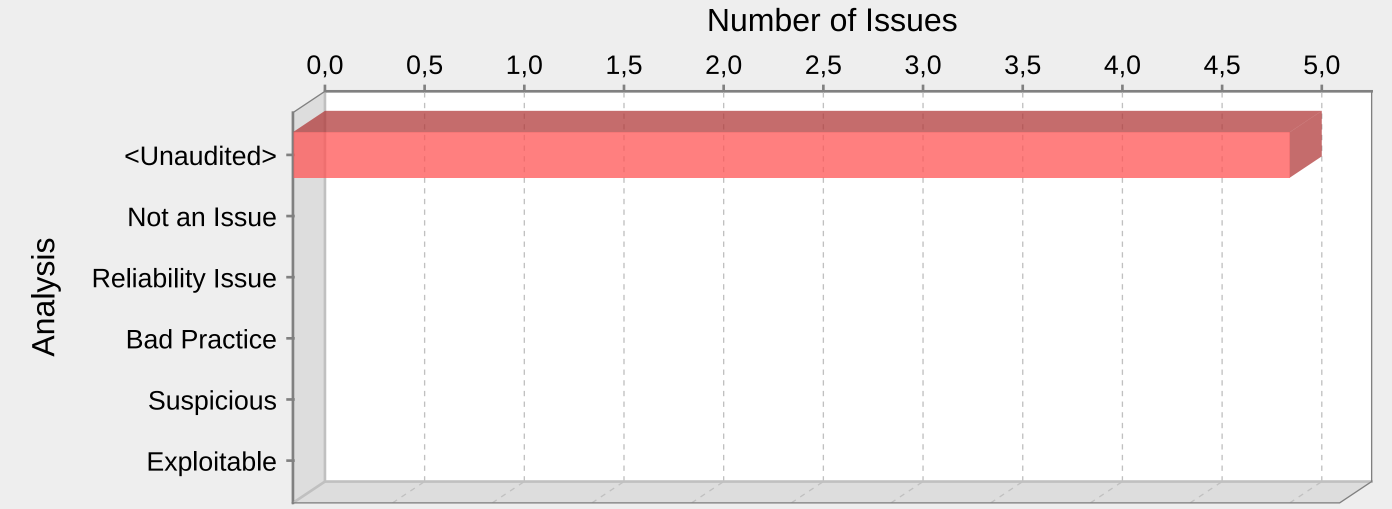
Fortify Priority:	Low	**Folder**	Low
Kingdom:	Security Features		
Abstract:	Null passwords can compromise security.		
Sink:	buffers.c:1554 FieldAccess: password()		

control.c, line 1079 (Password Management: Null Password)

Fortify Priority:	Low	**Folder**	Low
Kingdom:	Security Features		
Abstract:	Null passwords can compromise security.		

Sink:	control.c:1079 VariableAccess: password()		
test.c, line 249 (Password Management: Null Password)			
Fortify Priority:	Low	**Folder**	Low
Kingdom:	Security Features		
Abstract:	Null passwords can compromise security.		
Sink:	test.c:249 FieldAccess: password()		
control.c, line 1221 (Password Management: Null Password)			
Fortify Priority:	Low	**Folder**	Low
Kingdom:	Security Features		
Abstract:	Null passwords can compromise security.		
Sink:	control.c:1221 VariableAccess: password()		
control.c, line 1212 (Password Management: Null Password)			
Fortify Priority:	Low	**Folder**	Low
Kingdom:	Security Features		
Abstract:	Null passwords can compromise security.		
Sink:	control.c:1212 VariableAccess: password()		

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.

tinytest.c, line 97 (Type Mismatch: Signed to Unsigned)

Fortify Priority:	High	**Folder**	High
Kingdom:	Code Quality		
Abstract:	The function testcase_run_bare_() in tinytest.c is declared to return an unsigned value, but on line 97 it returns a signed value.		
Sink:	tinytest.c:97 ReturnStatement()		

tinytest.c, line 194 (Type Mismatch: Signed to Unsigned)

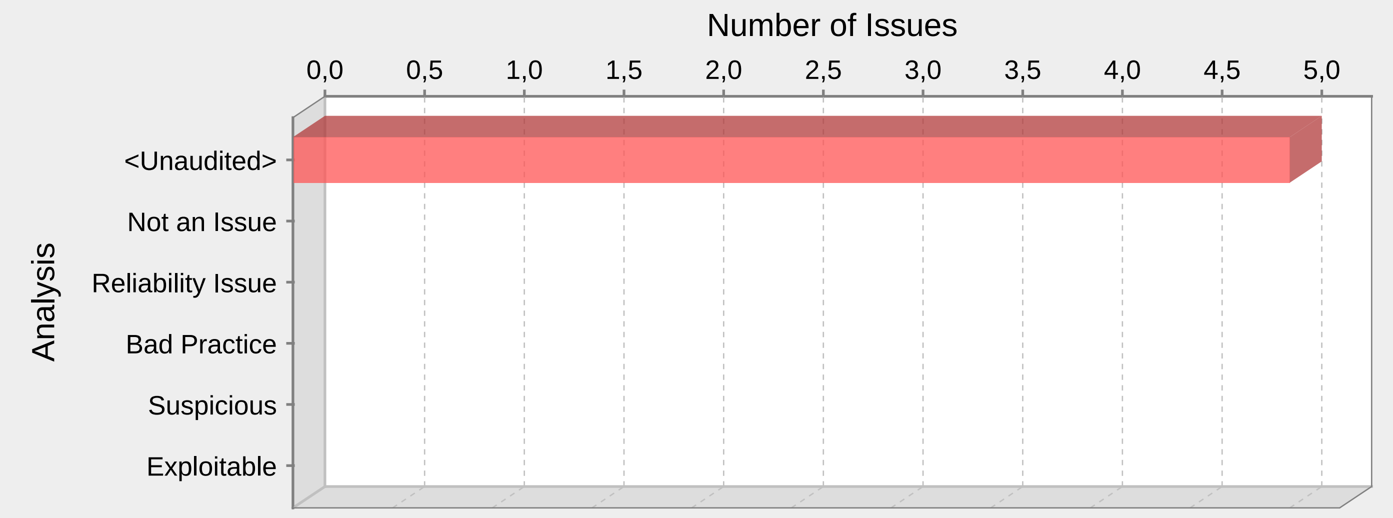
Fortify Priority:	High	**Folder**	High
Kingdom:	Code Quality		
Abstract:	The function testcase_run_forked_() in tinytest.c is declared to return an unsigned value, but on line 194 it returns a signed value.		
Sink:	tinytest.c:194 ReturnStatement()		

router.c, line 1285 (Type Mismatch: Signed to Unsigned)			
Fortify Priority:	High	**Folder**	High
Kingdom:	Code Quality		
Abstract:	The function router_get_advertised_dir_port() in router.c is declared to return an unsigned value, but on line 1285 it returns a signed value.		
Sink:	router.c:1285 ReturnStatement()		

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()		

router.c, line 1265 (Type Mismatch: Signed to Unsigned)			
Fortify Priority:	High	**Folder**	High
Kingdom:	Code Quality		
Abstract:	The function router_get_advertised_or_port() in router.c is declared to return an unsigned value, but on line 1265 it returns a signed value.		
Sink:	router.c:1265 ReturnStatement()		

Category: Uninitialized Variable (5 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:
aN = -1; aN = -1;
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.

Recommendations:

Before a variable is used, initialize it.

connection.c, line 1035 (Uninitialized Variable)

Fortify Priority:	Low	**Folder**	Low
Kingdom:	Code Quality		
Abstract:	The function connection_listener_new() in connection.c uses the variable s before it has been initialized.		
Sink:	connection.c:1035 set_socket_nonblocking(s) : Variable s used without being initialized()		

control.c, line 2657 (Uninitialized Variable)

Fortify Priority:	Low	**Folder**	Low
Kingdom:	Code Quality		
Abstract:	The function handle_control_attachstream() in control.c uses the variable exit_digest before it has been initialized.		
Sink:	control.c:2657 hex_str(exit_digest, ?) : Variable exit_digest used without being initialized()		

connection.c, line 1257 (Uninitialized Variable)

Fortify Priority:	Low	**Folder**	Low
Kingdom:	Code Quality		
Abstract:	The function connection_handle_listener_read() in connection.c uses the variable newconn before it has been initialized.		
Sink:	connection.c:1257 connection_add_impl(newconn, ?) : Variable newconn used without being initialized()		

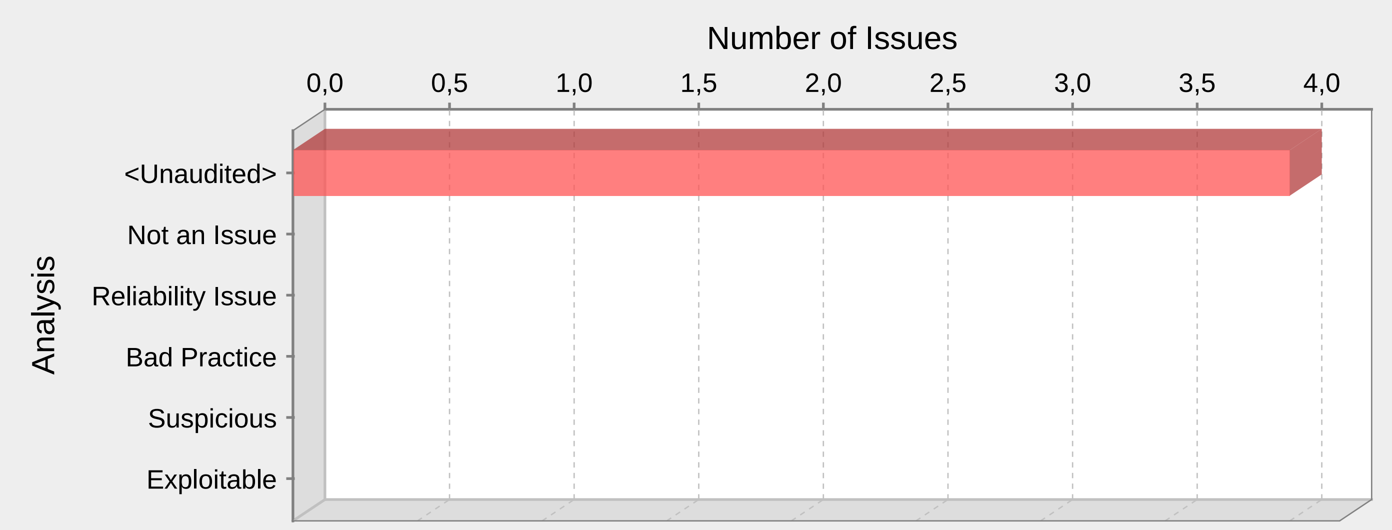
util.c, line 3718 (Uninitialized Variable)

Fortify Priority:	Low	**Folder**	Low
Kingdom:	Code Quality		
Abstract:	The function tor_spawn_background() in util.c uses the variable nbytes before it has been initialized.		
Sink:	util.c:3718 nbytes : Variable nbytes used without being initialized()		

aes.c, line 340 (Uninitialized Variable)

Fortify Priority:	Low	**Folder**	Low
Kingdom:	Code Quality		
Abstract:	The function aes_set_key() in aes.c uses the variable c before it has been initialized.		
Sink:	aes.c:340 c : Variable c used without being initialized()		

Category: Memory Leak: Reallocation (4 Issues)



Abstract:

The program resizes a block of allocated memory. If the resize fails, the original block will be leaked.

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 realloc() fails to resize the original allocation.

```
char* getBlocks(int fd) {
int amt;
int request = BLOCK_SIZE;
char* buf = (char*) malloc(BLOCK_SIZE + 1);
if (!buf) {
goto ERR;
}
amt = read(fd, buf, request);
while ((amt % BLOCK_SIZE) != 0) {
if (amt < request) {
goto ERR;
}
request = request + BLOCK_SIZE;
buf = realloc(buf, request);
if (!buf) {
goto ERR;
}
amt = read(fd, buf, request);
}

return buf;

ERR:
if (buf) {
free(buf);
}
return NULL;
}
```

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* getBlocks(int fd) {
int amt;
int request = BLOCK_SIZE;
char* newbuf;
char* buf = (char*) malloc(BLOCK_SIZE + 1);
if (!buf) {
goto ERR;
}
amt = read(fd, buf, request);
while ((amt % BLOCK_SIZE) != 0) {
if (amt < request) {
goto ERR;
}
request = request + BLOCK_SIZE;
newbuf = realloc(buf, request);
if (!newbuf) {
goto ERR;
}
buf = newbuf;
amt = read(fd, buf, request);
}

return buf;

ERR:
if (buf) {
free(buf);
}
if (newbuf) {
free(newbuf);
}
return NULL;
}
```

compat.c, line 1752 (Memory Leak: Reallocation)

Fortify Priority:	Low	**Folder**	Low
Kingdom:	Code Quality		
Abstract:	The function alloc_getcwd() in compat.c resizes a block of allocated memory on line 1752. If the resize fails, the original block will be leaked.		
Sink:	compat.c:1752 _tor_realloc(path, ?)		

test_util.c, line 1314 (Memory Leak: Reallocation)

Fortify Priority:	Low	**Folder**	Low
Kingdom:	Code Quality		
Abstract:	The function test_util_gzip() in test_util.c resizes a block of allocated memory on line 1314. If the resize fails, the original block will be leaked.		
Sink:	test_util.c:1314 _tor_realloc(buf2, ?)		

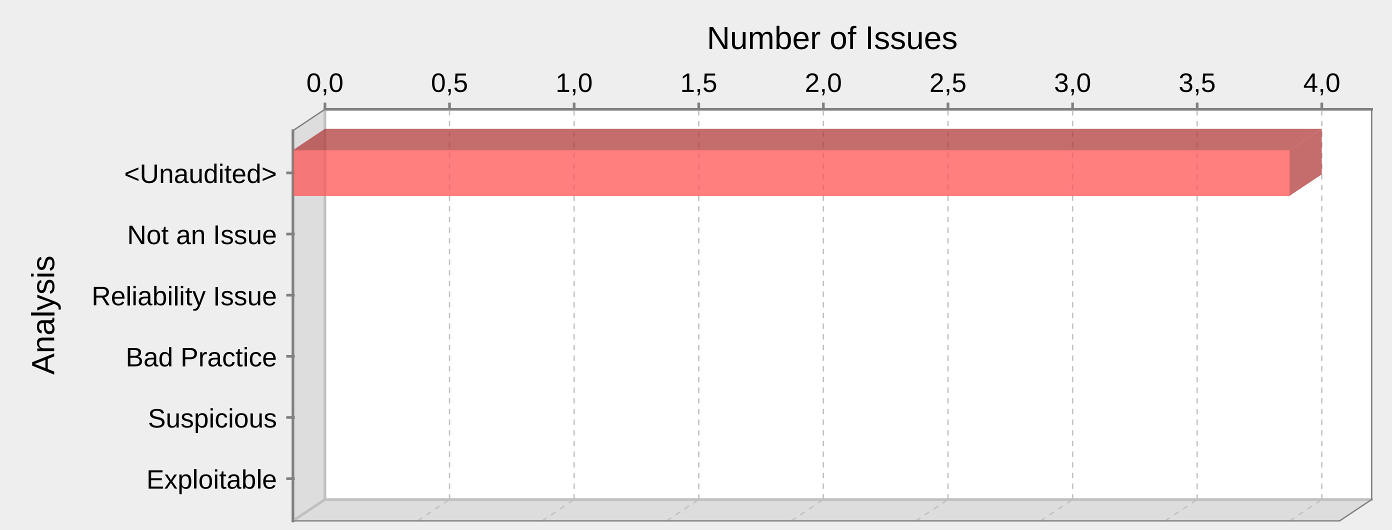
buffers.c, line 236 (Memory Leak: Reallocation)

Fortify Priority:	Low	**Folder**	Low

Kingdom:	Code Quality
Abstract:	The function chunk_grow() in buffers.c resizes a block of allocated memory on line 236. If the resize fails, the original block will be leaked.
Sink:	buffers.c:236 _tor_realloc(chunk, ?)

compat.c, line 1440 (Memory Leak: Reallocation)			
Fortify Priority:	Low	**Folder**	Low
Kingdom:	Code Quality		
Abstract:	The function log_credential_status() in compat.c resizes a block of allocated memory on line 1440. If the resize fails, the original block will be leaked.		
Sink:	compat.c:1440 _tor_realloc(sup_gids, ?)		

Category: System Information Leak (4 Issues)



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.

Recommendations:

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.

Tips:

1. Do not rely on wrapper scripts, corporate IT policy, or quick-thinking system administrators to prevent system information leaks. Write software that is secure on its own.

2. This category of vulnerability does not apply to all types of programs. For example, if your application executes on a client machine where system information is already available to an attacker, or if you print system information only to a trusted log file, you can use AuditGuide to filter out this category.

test_dir.c, line 169 (System Information Leak)

Fortify Priority:	Low	**Folder**	Low
Kingdom:	Encapsulation		
Abstract:	The function test_dir_formats() in test_dir.c reveals system data or debugging information by calling printf() on line 169. The information revealed by printf() could help an adversary form a plan of attack.		
Source:	compat.c:2102 uname()		
Sink:	test_dir.c:169 printf()		

test.c, line 150 (System Information Leak)

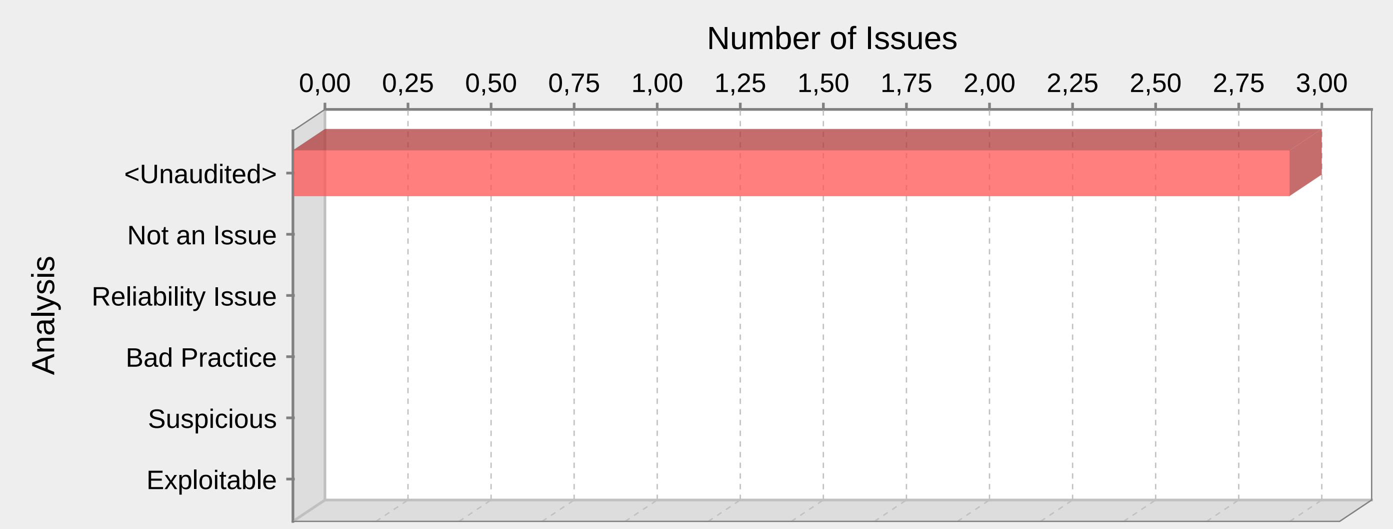
Fortify Priority:	Low	**Folder**	Low

Kingdom:	Encapsulation		
Abstract:	The function rm_rf() in test.c reveals system data or debugging information by calling fprintf() on line 150. The information revealed by fprintf() could help an adversary form a plan of attack.		
Source:	test.c:150 strerror()		
Sink:	test.c:150 fprintf()		

test_dir.c, line 169 (System Information Leak)			
Fortify Priority:	Low	**Folder**	Low
Kingdom:	Encapsulation		
Abstract:	The function test_dir_formats() in test_dir.c reveals system data or debugging information by calling printf() on line 169. The information revealed by printf() could help an adversary form a plan of attack.		
Source:	compat.c:2102 uname()		
Sink:	test_dir.c:169 printf()		

test.c, line 141 (System Information Leak)			
Fortify Priority:	Low	**Folder**	Low
Kingdom:	Encapsulation		
Abstract:	The function rm_rf() in test.c reveals system data or debugging information by calling fprintf() on line 141. The information revealed by fprintf() could help an adversary form a plan of attack.		
Source:	test.c:141 strerror()		
Sink:	test.c:141 fprintf()		

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:1773 read()		
Sink:	util.c:1771 recv()		

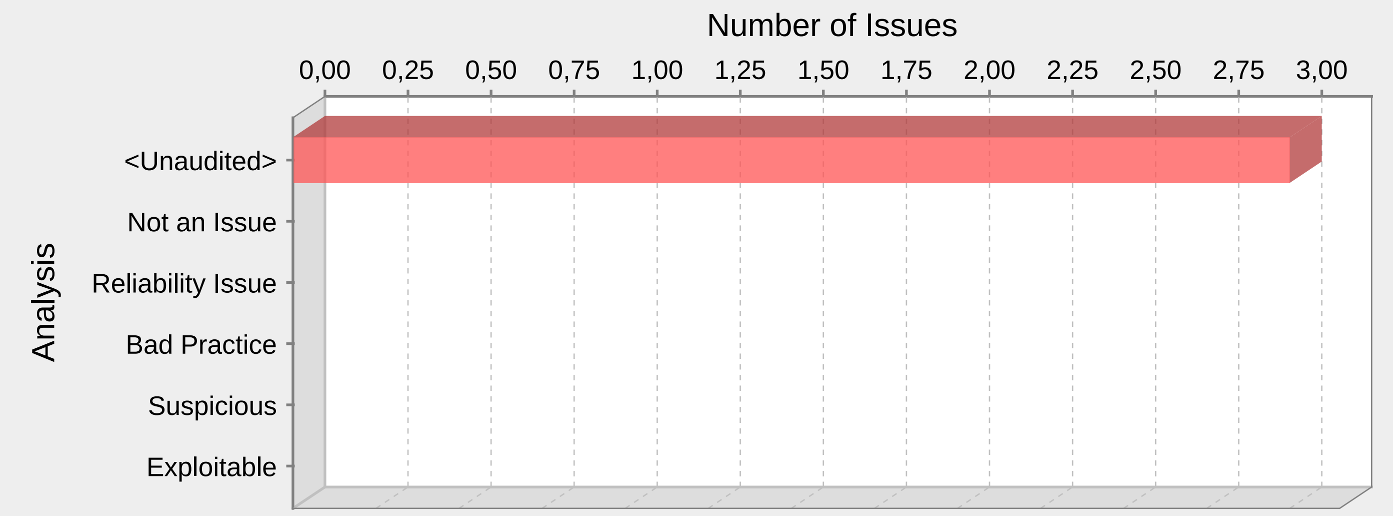
util.c, line 1771 (Buffer Overflow: Off-by-One)

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()		
Sink:	util.c:1771 recv()		

util.c, line 3347 (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.		
Sink:	util.c:3347 Assignment()		

Category: Format String (3 Issues)



Abstract:

Allowing an attacker to control a function's format string can result in a buffer overflow.

Explanation:

Format string vulnerabilities occur when an attacker is allowed to control the format string argument to a function like sprintf(), FormatMessageW(), or syslog().

Example 1: The following code copies a command line argument into a buffer using snprintf().

```
int main(int argc, char **argv){
char buf[128];
...
snprintf(buf,128,argv[1]);
}
```

This code allows an attacker to view the contents of the stack and write to the stack using a command line argument containing a sequence of formatting directives. The attacker can read from the stack by providing more formatting directives, such as %x, than the function takes as arguments to be formatted. (In this example, the function takes no arguments to be formatted.) By using the %n formatting directive, the attacker can write to the stack, causing snprintf() to write the number of bytes output thus far to the specified argument (rather than reading a value from the argument, which is the intended behavior). A sophisticated version of this attack will use four staggered writes to completely control the value of a pointer on the stack.

Example 2: Certain implementations make more advanced attacks even easier by providing format directives that control the location in memory to read from or write to. An example of these directives is shown in the following code, written for glibc:

```
printf("%d %d %1$d %1$d\n", 5, 9);
```

This code produces the following output:

```
5 9 5 5
```

It is also possible to use half-writes (%hn) to accurately control arbitrary DWORDS in memory, which greatly reduces the complexity needed to execute an attack that would otherwise require four staggered writes, such as the one mentioned in Example 1.

Example 3: Simple format string vulnerabilities often result from seemingly innocuous shortcuts. The use of some such shortcuts is so ingrained that programmers might not even realize that the function they are using expects a format string argument.

For example, the syslog() function is sometimes used as follows:

```
...
syslog(LOG_ERR, cmdBuf);
...
```

Because the second parameter to syslog() is a format string, any formatting directives included in cmdBuf are interpreted as described in Example 1.

The following code shows a correct usage of syslog():

```
...
```

```
syslog(LOG_ERR, "%s", cmdBuf);
...
```

Recommendations:

Whenever possible, pass static format strings to functions that accept a format string argument. If format strings must be constructed dynamically, define a set of valid format strings and make selections from this safe set. Finally, always verify that the number of formatting directives in the selected format string corresponds to the number of arguments to be formatted.

compat.c, line 422 (Format String)

Fortify Priority:	Low	**Folder**	Low
Kingdom:	Input Validation and Representation		
Abstract:	An attacker might be able to control the format string argument to vasprintf() at compat.c line 422, allowing an attack much like a buffer overflow.		
Sink:	compat.c:422 vasprintf(1)		

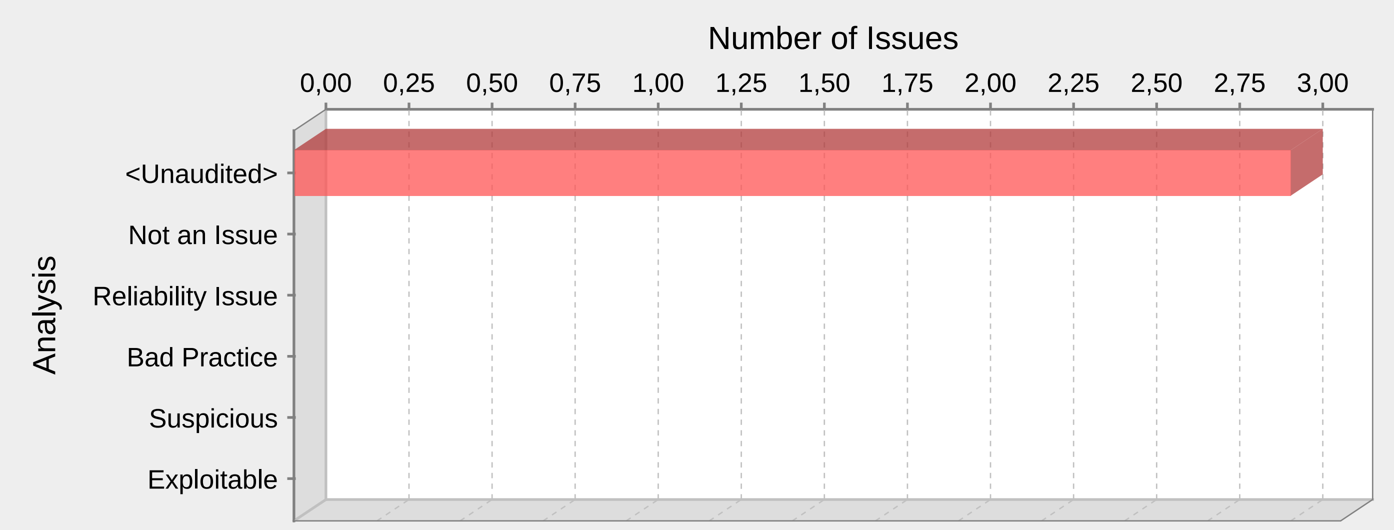
compat.c, line 372 (Format String)

Fortify Priority:	Low	**Folder**	Low
Kingdom:	Input Validation and Representation		
Abstract:	An attacker might be able to control the format string argument to vsnprintf() at compat.c line 372, allowing an attack much like a buffer overflow.		
Sink:	compat.c:372 vsnprintf(2)		

eventdns.c, line 438 (Format String)

Fortify Priority:	Low	**Folder**	Low
Kingdom:	Input Validation and Representation		
Abstract:	An attacker might be able to control the format string argument to vsnprintf() at eventdns.c line 438, allowing an attack much like a buffer overflow.		
Sink:	eventdns.c:438 vsnprintf(2)		

Category: Weak Cryptographic Hash (3 Issues)



Abstract:

Weak cryptographic hashes cannot guarantee data integrity and should not be used in security-critical contexts.

Explanation:

MD5 and SHA-1 are popular cryptographic hash algorithms often used to verify the integrity of messages and other data. Recent advances in cryptanalysis have discovered weaknesses in both algorithms. Consequently, MD5 and SHA-1 should no longer be relied upon to verify the authenticity of data in security-critical contexts.

Techniques for breaking MD5 hashes are advanced and widely available enough that the algorithm must not be relied upon for security. In the case of SHA-1, current techniques still require a significant amount of computational power and are more difficult to implement. However, attackers have found the Achilles' heel for the algorithm, and techniques for breaking it will likely lead to the discovery of even faster attacks.

Recommendations:

Discontinue the use of MD5 and SHA-1 for data-verification in security-critical contexts. Currently, SHA-224, SHA-256, SHA-384 and SHA-512 are good alternatives. However, these variants of the Secure Hash Algorithm are not standardized and have not been scrutinized as closely as SHA-1, so be mindful of future research that might impact the security of these algorithms.

tortls.c, line 634 (Weak Cryptographic Hash)

Fortify Priority:	Low	**Folder**	Low
Kingdom:	Security Features		
Abstract:	Weak cryptographic hashes cannot guarantee data integrity and should not be used in security-critical contexts.		
Sink:	tortls.c:634 EVP_sha1()		

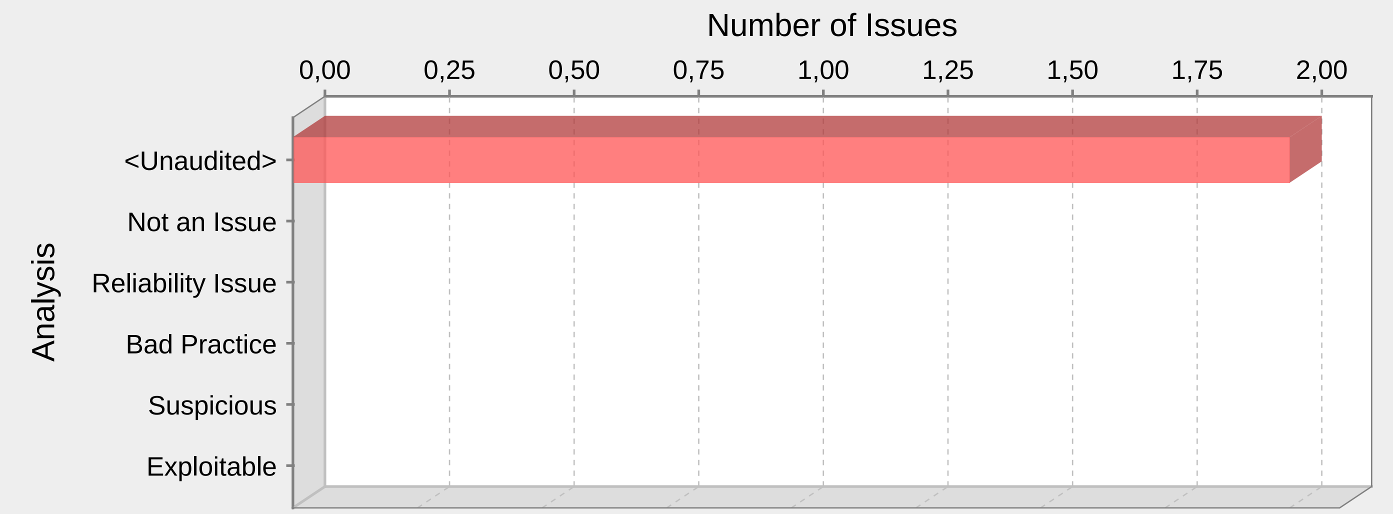
crypto.c, line 1487 (Weak Cryptographic Hash)

Fortify Priority:	Low	**Folder**	Low
Kingdom:	Security Features		
Abstract:	Weak cryptographic hashes cannot guarantee data integrity and should not be used in security-critical contexts.		
Sink:	crypto.c:1487 SHA1_Init()		

crypto.c, line 1613 (Weak Cryptographic Hash)

Fortify Priority:	Low	**Folder**	Low
Kingdom:	Security Features		
Abstract:	Weak cryptographic hashes cannot guarantee data integrity and should not be used in security-critical contexts.		
Sink:	crypto.c:1613 EVP_sha1()		

Category: Command Injection (2 Issues)



Abstract:

Executing commands without specifying an absolute path could allow an attacker to execute a malicious binary by changing $PATH or other aspects of the program's execution environment.

Explanation:

Command injection vulnerabilities take two forms:

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

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

In this case we are primarily concerned with the second scenario, in which an attacker can change the meaning of the command by changing an environment variable or by inserting a malicious executable early on the search path. Command injection vulnerabilities of this type occur when:

1. An attacker modifies an application's environment.

2. The application executes a command without specifying an absolute path or verifying the binary being executed.

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

Example 1: This example demonstrates what can happen when the attacker can change how a command is interpreted. The code is from a web-based CGI utility that allows users to change their passwords. The password update process under NIS includes running make in the /var/yp directory. Note that since the program updates password records, it has been installed setuid root.

The program invokes make as follows:

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

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

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

Recommendations:

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

Be aware that the Windows APIs impose a specific search order that is based not only on a series of directories, but also on a list of file extensions that are automatically appended if none is specified.

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

util.c, line 3696 (Command Injection)

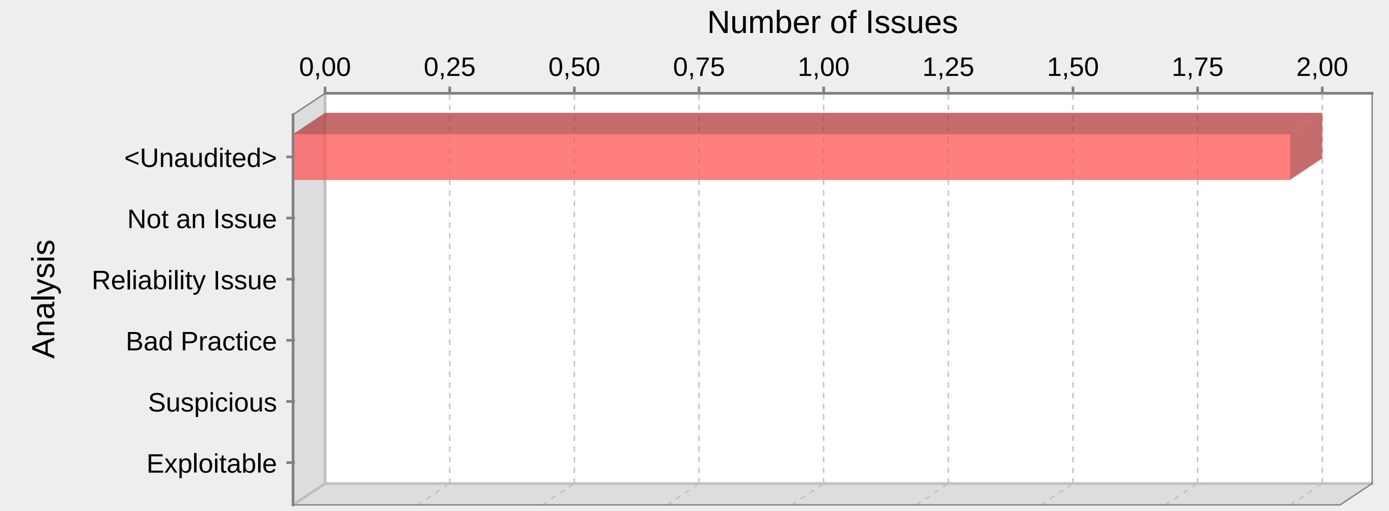
Fortify Priority:	Low	**Folder**	Low

Kingdom:	Input Validation and Representation
Abstract:	The function tor_spawn_background() in util.c calls execvp() on line 3696 to execute a command. This might allow an attacker to inject malicious commands.
Sink:	util.c:3696 execvp(0)

util.c, line 3694 (Command Injection)

Fortify Priority:	Low	**Folder**	Low
Kingdom:	Input Validation and Representation		
Abstract:	The function tor_spawn_background() in util.c calls execve() on line 3694 to execute a command. This might allow an attacker to inject malicious commands.		
Sink:	util.c:3694 execve()		

Category: Dead Code (2 Issues)



Abstract:

This statement will never be executed.

Explanation:

The surrounding code makes it impossible for this statement to ever be executed.

Example: The condition for the second if statement is impossible to satisfy. It requires that the variable s be non-null, while on the only path where s can be assigned a non-null value there is a return statement.

```
String s = null;
if (b) {
s = "Yes";
return;
}
if (s != null) {
Dead();
}
```

Recommendations:

In general, you should repair or remove unused code. It causes additional complexity and maintenance burden without contributing to the functionality of the program.

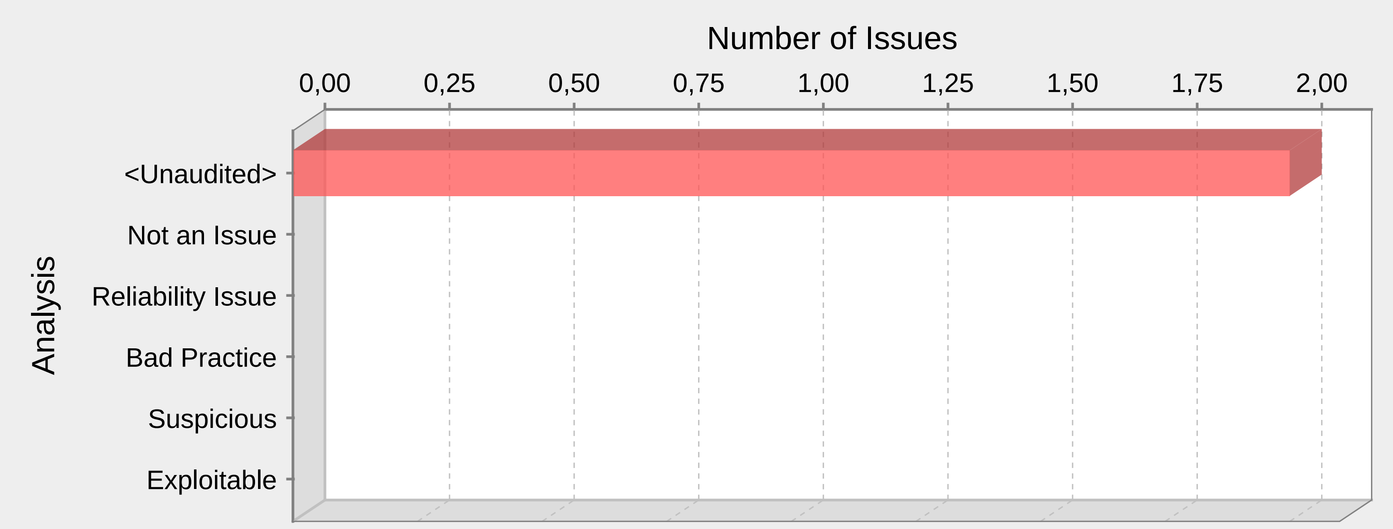
dns.c, line 1454 (Dead Code)

Fortify Priority:	Low	**Folder**	Low
Kingdom:	Code Quality		
Abstract:	Line 1454 in dns.c will never be executed. It is dead code.		
Sink:	dns.c:1454 null()		

compat_libevent.c, line 460 (Dead Code)

Fortify Priority:	Low	**Folder**	Low
Kingdom:	Code Quality		
Abstract:	Line 460 in compat_libevent.c will never be executed. It is dead code.		
Sink:	compat_libevent.c:460 null()		

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].

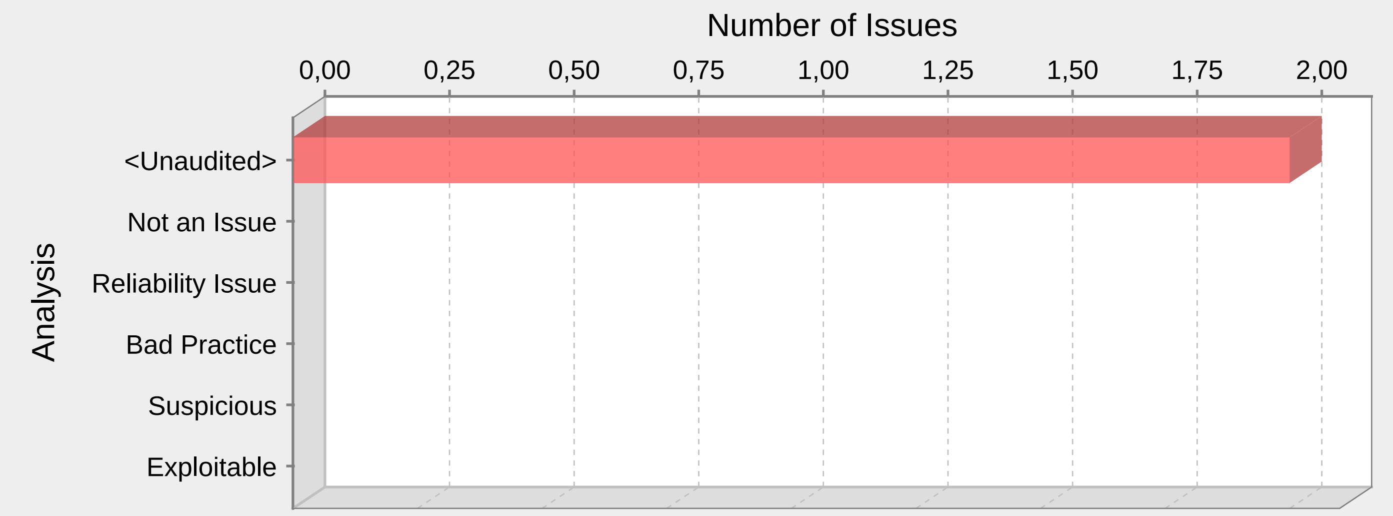
compat.c, line 2083 (Insecure Randomness)

Fortify Priority:	High	**Folder**	High
Kingdom:	Security Features		
Abstract:	The random number generator implemented by random() cannot withstand a cryptographic attack.		
Sink:	compat.c:2083 random()		

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()		

Category: 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);
```

Recommendations:

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

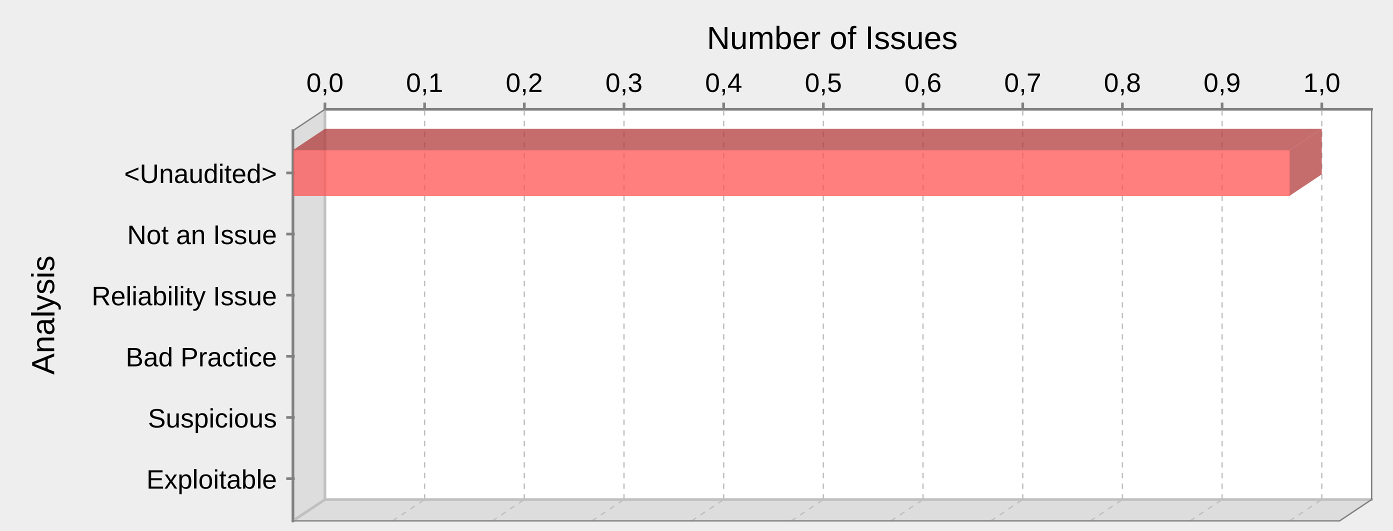
routerlist.c, line 1710 (Poor Style: Redundant Initialization)

Fortify Priority:	Low	**Folder**	Low
Kingdom:	Code Quality		
Abstract:	The function smartlist_choose_node_by_bandwidth_weights() in routerlist.c never uses the initial value it assigns to the variable tmp on line 1710.		
Sink:	routerlist.c:1710 VariableAccess: tmp()		

circuitbuild.c, line 909 (Poor Style: Redundant Initialization)

Fortify Priority:	Low	**Folder**	Low
Kingdom:	Code Quality		
Abstract:	The function circuit_build_times_filter_timeouts() in circuitbuild.c never uses the initial value it assigns to the variable timeout_rate on line 909.		
Sink:	circuitbuild.c:909 VariableAccess: timeout_rate()		

Category: Insecure Compiler Optimization (1 Issues)



Abstract:

Improperly scrubbing sensitive data from memory can compromise security.

Explanation:

Compiler optimization errors occur when:

1. Secret data is stored in memory.

2. The secret data is scrubbed from memory by overwriting its contents.

3. The source code is compiled using an optimizing compiler, which identifies and removes the function that overwrites the contents as a dead store because the memory is not used subsequently.

Example: The following code reads a password from the user, uses the password to connect to a back-end mainframe and then attempts to scrub the password from memory using memset().

```
void GetData(char *MFAddr) {
char pwd[64];
if (GetPasswordFromUser(pwd, sizeof(pwd))) {
if (ConnectToMainframe(MFAddr, pwd)) {
// Interaction with mainframe
}
}
memset(pwd, 0, sizeof(pwd));
}
```

The code in the example will behave correctly if it is executed verbatim, but if the code is compiled using an optimizing compiler, such as Microsoft Visual C++(R) .NET or GCC 3.x, then the call to memset() will be removed as a dead store because the buffer pwd is not used after its value is overwritten [2]. Because the buffer pwd contains a sensitive value, the application may be vulnerable to attack if the data is left memory resident. If attackers are able to access the correct region of memory, they may use the recovered password to gain control of the system.

It is common practice to overwrite sensitive data manipulated in memory, such as passwords or cryptographic keys, in order to prevent attackers from learning system secrets. However, with the advent of optimizing compilers, programs do not always behave as their source code alone would suggest. In the example, the compiler interprets the call to memset() as dead code because the memory being written to is not subsequently used, despite the fact that there is clearly a security motivation for the operation to occur. The problem here is that many compilers, and in fact many programming languages, do not take this and other security concerns into consideration in their efforts to improve efficiency.

Attackers typically exploit this type of vulnerability by using a core dump or runtime mechanism to access the memory used by a particular application and recover the secret information. Once an attacker has access to the secret information, it is relatively straightforward to further exploit the system and possibly compromise other resources with which the application interacts.

Recommendations:

Optimizing compilers are hugely beneficial to performance, so disabling optimization is rarely a reasonable option. The solution is to communicate to the compiler exactly how the program should behave. Because support for this communication is imperfect and varies from platform to platform, current solutions to the problem are imperfect as well.

It is often possible to force the compiler into retaining calls to scrubbing functions by reading from the variable after it is cleaned in memory. Another option involves volatile pointers, which are not currently optimized because they can be modified from outside the application. You can make use of this fact to trick the compiler by casting pointers to sensitive data to volatile pointers. This could be accomplished in the example above by adding the following line immediately after the call to memset():

```
*(volatile char*)pwd = *(volatile char*)pwd;
```

Although both of these solutions prevent existing compilers from optimizing out calls to scrubbing functions like the one seen in the example above, they rely on current optimization techniques, which will continue to evolve in the future. The insidious aspect of this is that, as compiler technology evolves, security flaws like this one may be reintroduced even if an application's source code has remained unchanged.

On recent Windows(R) platforms, consider using SecureZeroMemory(), which is a secure replacement for ZeroMemory() that uses the volatile pointer trick given above to protect itself from optimization [2]. Additionally, in most versions of Microsoft Visual C++(R) it is possible to use the #pragma optimize construct to prevent the compiler from optimizing specific blocks of code. For example:

```
#pragma optimize("",off);
memset(pwd, 0, sizeof(pwd));
#pragma optimize("",on);
```

Tips:

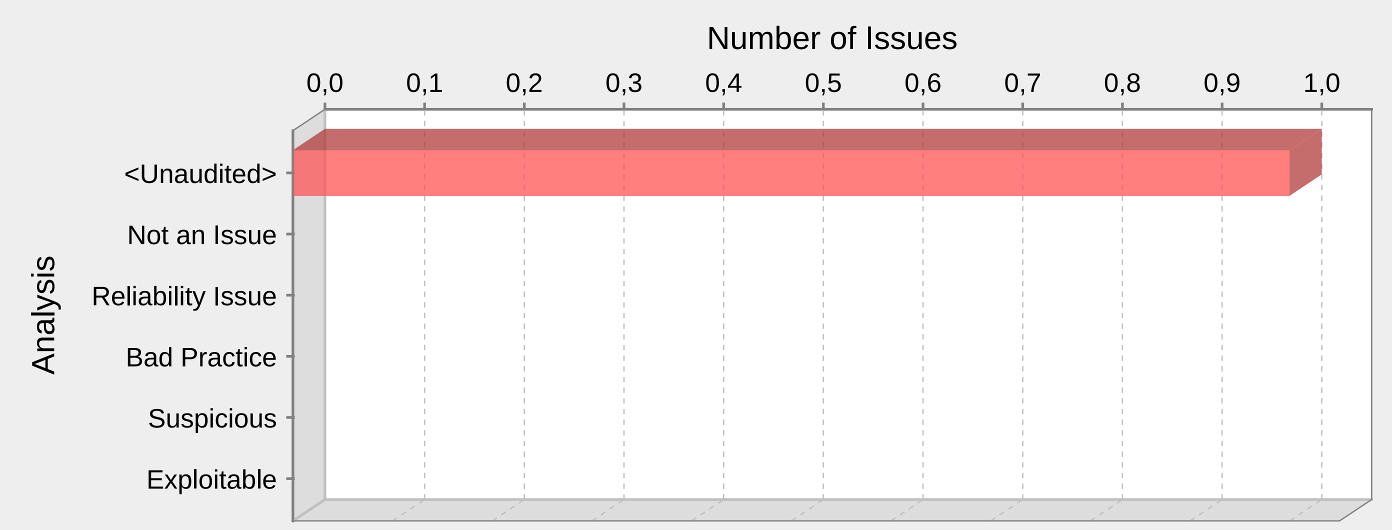
1. The HP Fortify Secure Coding Rulepacks will identify uses of memset() that could potentially be removed by an optimizing compiler. HP Fortify does not flag all calls to memset(), only calls where the memory being scrubbed is not referenced after the call.

In order to audit this issue, you need to determine whether or not the memory contains sensitive information.

connection.c, line 1163 (Insecure Compiler Optimization)

Fortify Priority:	Low	**Folder**	Low
Kingdom:	Environment		
Abstract:	If calling memset() is intended to scrub sensitive data from memory, it might fail to achieve the intended effect.		
Sink:	connection.c:1163 memset()		

Category: Out-of-Bounds Read (1 Issues)



Abstract:

The program reads data from outside the bounds of allocated memory.

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, the program reads from outside the bounds of allocated memory, which can allow access to sensitive information, introduce incorrect behavior, or cause the program to crash.

Example 1: In the following code, the call to memcpy() reads memory from outside the allocated bounds of cArray, which contains MAX elements of type char, while iArray contains MAX elements of type int.

```
void MemFuncs() {
char array1[MAX];
int  array2[MAX];
memcpy(array2, array1, sizeof(array2));
}
```

Example 2: The following short program uses an untrusted command line argument as the search buffer in a call to memchr() with a constant number of bytes to be analyzed.

```
int main(int argc, char** argv) {
char* ret = memchr(argv[0], 'x', MAX_PATH);
printf("%s\n", ret);
}
```

The program is meant to print a substring of argv[0], by searching the argv[0] data up to a constant number of bytes. However, as the (constant) number of bytes might be larger than the data allocated for argv[0], the search might go on beyond the data allocated for argv[0]. This will be the case when x is not found in argv[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.

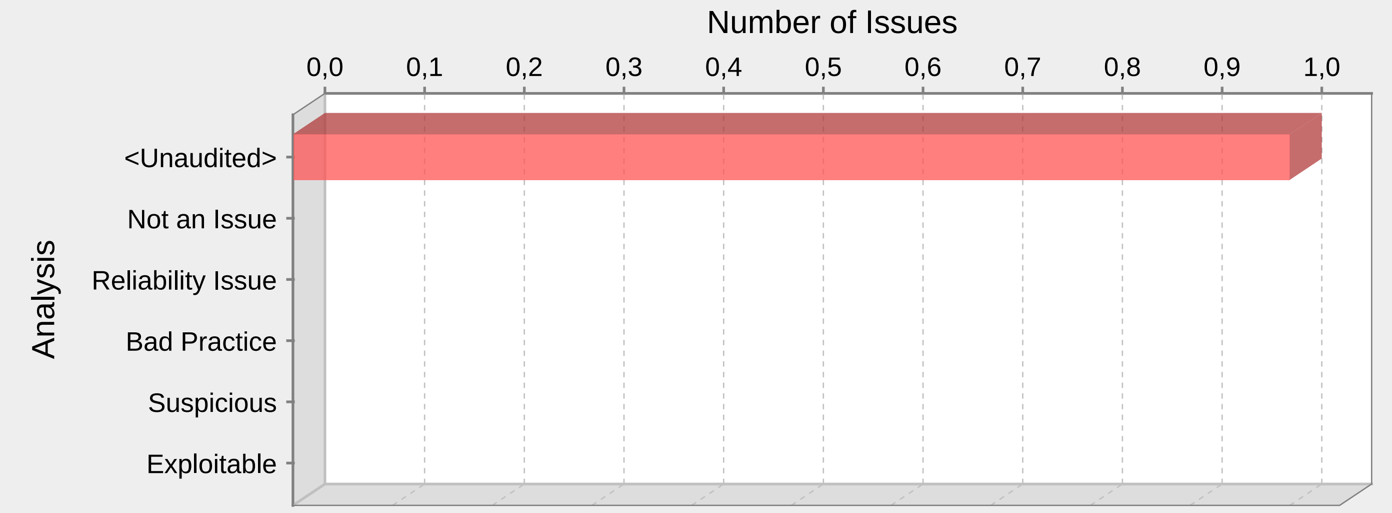
- When input data is too large, be leery of truncating the data and continuing to process it. Truncation can change the meaning of the input.

- 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.

dirserv.c, line 3400 (Out-of-Bounds Read)

Fortify Priority:	Low	**Folder**	Low
Kingdom:	Input Validation and Representation		
Abstract:	The program reads data from outside the bounds of allocated memory.		
Sink:	dirserv.c:3400 FunctionCall: memchr()		

Category: Out-of-Bounds Read: Off-by-One (1 Issues)



Abstract:

The program reads data from just outside the bounds of allocated memory.

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.

In this case, the program reads from outside the bounds of allocated memory, which can allow access to sensitive information, introduce incorrect behavior, or cause the program to crash.

Example: The following code sequentially dereferences five-element array of char, with the last reference introducing an off-by-one error.

```
char Read() {

char buf[5];
return 0
+ buf[0]
+ buf[1]
+ buf[2]
+ buf[3]
+ buf[4]
+ buf[5];
}
```

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.

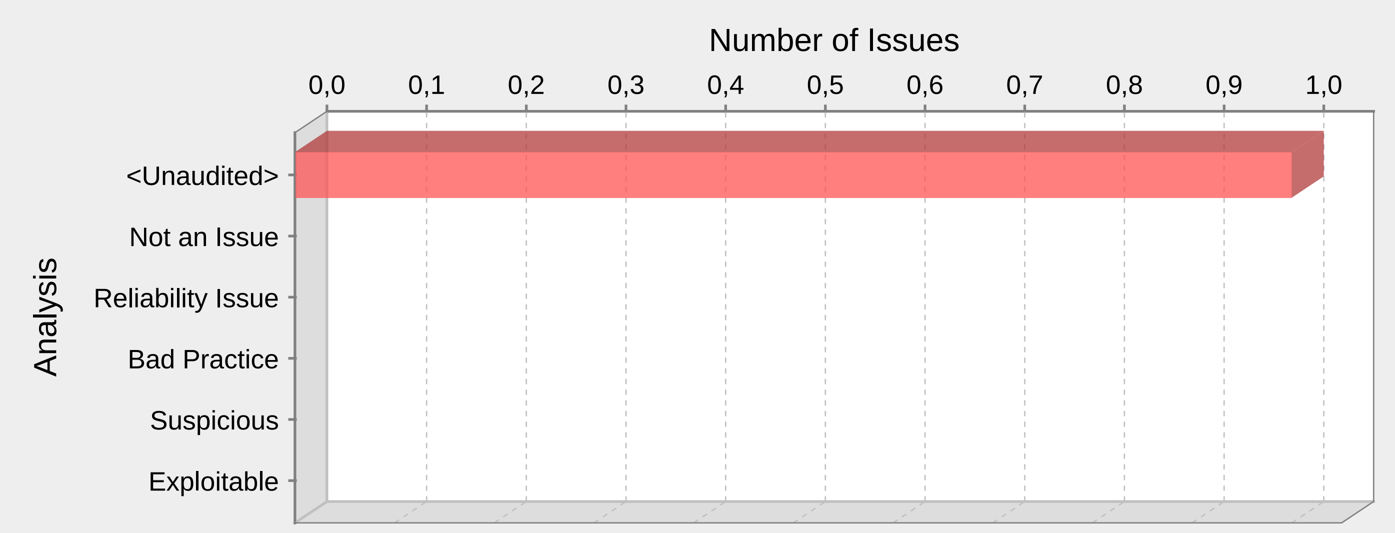
- When input data is too large, be leery of truncating the data and continuing to process it. Truncation can change the meaning of the input.

- 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.

crypto.c, line 1278 (Out-of-Bounds Read: Off-by-One)

Fortify Priority:	Low	**Folder**	Low
Kingdom:	Input Validation and Representation		
Abstract:	The program reads data from just outside the bounds of allocated memory.		
Sink:	crypto.c:1278 Read()		

Category: Portability Flaw (1 Issues)



Abstract:

Functions with inconsistent implementations across operating systems and operating system versions cause portability problems.

Explanation:

The behavior of functions in this category varies by operating system, and at times, even by operating system version. Implementation differences can include:

- Slight differences in the way parameters are interpreted leading to inconsistent results.
- Some implementations of the function carry significant security risks.
- The function might not be defined on all platforms.

Recommendations:

In order to write portable code, avoid using functions with inconsistent implementations.

In some cases the problem with a function can be mitigated. For example, if you perform adequate input validation, you can guard against internal buffer overflows in calls to getopt().

Usually, however, you cannot avoid problems when using certain functions. Examples include:

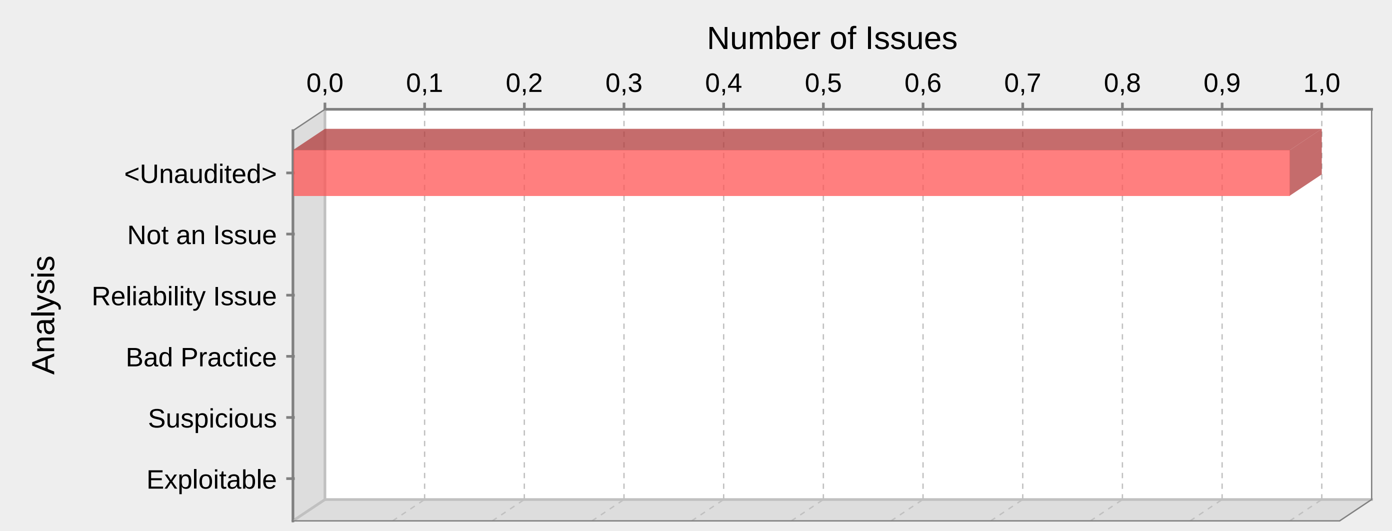
- vfork() implementations vary from platform to platform.
- strcmpi() is not defined on some Unix systems.
- memmem() is problematic due to changes between versions whereby the order of the arguments is reversed.

In general, replace calls to functions that are implemented inconsistently with safer counterparts.

compat.c, line 488 (Portability Flaw)

Fortify Priority:	Low	**Folder**	Low
Kingdom:	Code Quality		
Abstract:	The call to memmem() on line 488 causes portability problems because it has different semantics under different operating systems and operating system versions.		
Sink:	compat.c:488 memmem()		

Category: Resource Injection (1 Issues)



Abstract:

Allowing user input to control resource identifiers could enable an attacker to access or modify otherwise protected system resources.

Explanation:

A resource injection issue occurs when the following two conditions are met:

1. An attacker can specify the identifier used to access a system resource.

For example, an attacker might be able to specify a port number to be used to connect to a network resource.

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 transmit sensitive information to a third-party server.

Note: Resource injection that involves resources stored on the filesystem goes by the name path manipulation and is reported in separate category. See the path manipulation description for further details of this vulnerability.

Example: The following code uses a port number read from a CGI request to create a socket.

```
...
char* rPort = getenv("rPort");
...
serv_addr.sin_port = htons(atoi(rPort));
if (connect(sockfd,&serv_addr,sizeof(serv_addr)) < 0)
error("ERROR connecting");
...
```

The kind of resource affected by user input indicates the kind of content that may be dangerous. For example, data containing special characters like period, slash, and backslash are risky when used in methods that interact with the file system. Similarly, data that contains URLs and URIs is risky for functions that create remote connections.

Recommendations:

The best way to prevent resource injection 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.

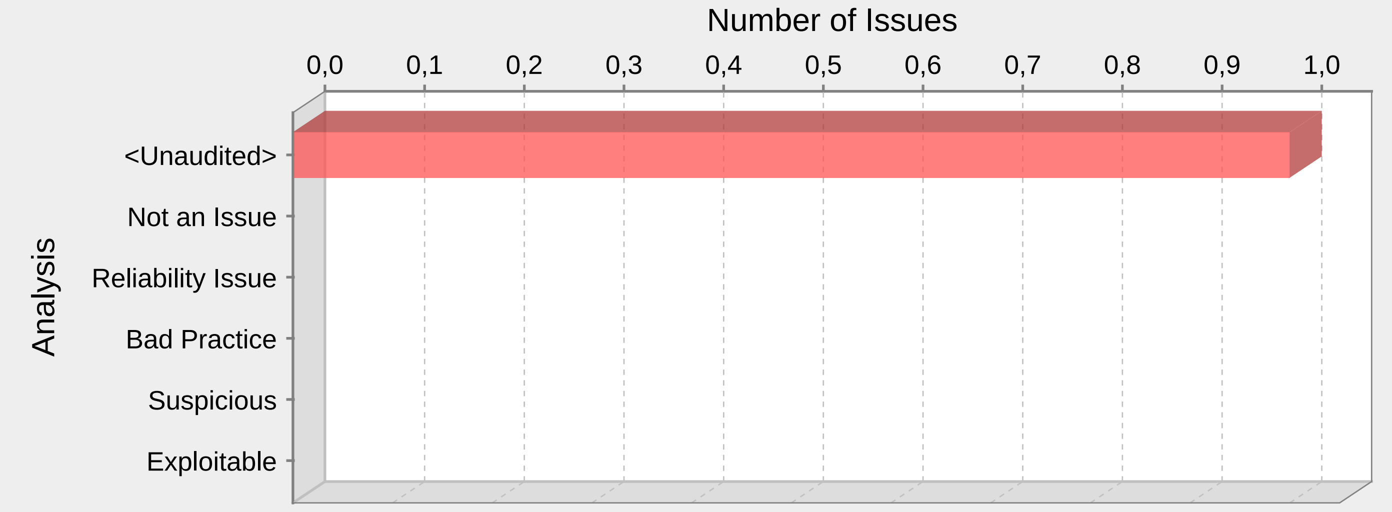
Tips:

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.

tor-resolve.c, line 206 (Resource Injection)			
Fortify Priority:	Low	**Folder**	Low
Kingdom:	Input Validation and Representation		
Abstract:	Attackers can control the resource identifier argument to connect() at tor-resolve.c line 206, which could enable them to access or modify otherwise protected system resources.		
Source:	tor-resolve.c:318 main(1)		
Sink:	tor-resolve.c:206 connect()		

Category: Setting Manipulation (1 Issues)



Abstract:

Allowing external control of system settings can disrupt service or cause an application to behave in unexpected ways.

Explanation:

Setting manipulation vulnerabilities occur when an attacker can control values that govern the behavior of the system, manage specific resources, or in some way affect the functionality of the application.

Because setting manipulation covers a diverse set of functions, any attempt at illustrating it will inevitably be incomplete. Rather than searching for a tight-knit relationship between the functions addressed in the setting manipulation category, take a step back and consider the sorts of system values that an attacker should not be allowed to control.

Example 1: The following C code accepts a number as one of its command line parameters and sets it as the host ID of the current machine.

```
...
sethostid(argv[1]);
...
```

Although a process must be privileged to successfully invoke sethostid(), unprivileged users may be able to invoke the program. The code in this example allows user input to directly control the value of a system setting. If an attacker provides a malicious value for host ID, the attacker can misidentify the affected machine on the network or cause other unintended behavior.

In general, do not allow user-provided or otherwise untrusted data to control sensitive values. The leverage that an attacker gains by controlling these values is not always immediately obvious, but do not underestimate the creativity of your attacker.

Recommendations:

Do not allow untrusted data to control sensitive values. In many cases where this error occurs, the application expects a particular input to hold only a very small range of values. If possible, instead of relying on the input to remain within an expected range, the application should guarantee reasonable behavior by using the input only to select from a predetermined set of safe values. If the input is maliciously crafted, the value passed to the sensitive function should default to some safe selection from this set. Even if the set of safe values cannot be known in advance, it is often possible to validate that the input falls within some safe range of values. If neither of these forms of validation is possible, you may have to redesign the application to avoid the need to accept potentially dangerous values from the user.

Tips:

1. You do not need to find a "smoking gun" situation in which the ability of an attacker to control some value leads to a potential exploit. The ideal mindset for auditing setting manipulation issues is to ask the questions: "Why is the user allowed to control this value? What could possibly happen?"

compat.c, line 1508 (Setting Manipulation)

Fortify Priority:	Low	**Folder**	Low
Kingdom:	Input Validation and Representation		
Abstract:	An attacker can control one of the the arguments to setgroups() at compat.c line 1508, which can lead to a disruption of service or unexpected application behavior.		
Source:	compat.c:1501 getpwnam()		
Sink:	compat.c:1508 setgroups()		