Security impact of *noexcept*

Project: C++ Programming Language – Core Working Group

Document Number: N3103-10-0093

Date: 2010-08-23

Authors: David Kohlbrenner, David Svoboda, and Andrew Wesie

Abstract

In this paper, we demonstrate that if a function marked **`noexcept`** actually throws an exception, and such behavior does not cause immediate termination, then the program can produce unexpected and counter-intuitive behavior. Furthermore, the behavior of such a program can be exploited by a malicious user to bypass security restrictions and cause denial-of-service attacks. Because noexcept has already been admitted to the C++0x draft standard, removing it is not an option. We therefore support the requirement that an exception thrown from a function marked 'noexcept' should immediately terminate the program.

Background

To understand **`noexcept`**, we begin with a small code example, compliant with C++2003:

```
void f();   // defined in some library (source not accessible)
void g() throw() { f(); }
```

This example will call **`std::unexpected()`** when **`f()`** throws an exception.

The **`noexcept`** feature serves as a partial replacement to the throws() declaration when a function claims not to throw any exceptions. ISO/IEC JTC1 SC22 WG21 N3092, C++ Final Committee Draft) Section 15.5.1, "The **`std::terminate()`** function" states that exception handling must be abandoned for less subtle error handling techniques:

when the search for a handler (15.3) encounters the outermost block of a function with a noexcept-specification that does not allow the exception (15.4)

The controversy arises if the **`throw()`** declaration is replaced with **`noexcept(true)`** , and **`f()`** throws an exception. Before the WG21 meeting in Pittsburgh[iii], an exception thrown by **`f()`** would result in undefined behavior. At the Pittsburgh meeting, it was decided that the program should call **`std::terminate()`** instead. However, the issue is not considered resolved, and it is possible the committee will decide that violation of noexcept should, once again, yield undefined behavior.

Ramifications

In this document, we present one code example to illustrate the potential dangers of **`noexcept`**, and to defend the invocation of **`std::terminate()`** in the event of a violation. This example illustrates how failure to terminate could be used to bypass a security check, and consequently grant privileged functionality to an unauthorized user.

Because an implementation of **`noexcept`** is not publicly available, we simulate the proposed behavior of **`noexcept`** using the *Microsoft Visual Studio 2010* compiler. In the *MSVC 2010* implementation, when an exception is thrown from a method with an empty exception specification, the program does not unwind the stack and does not call **`std::terminate()`**, but instead invokes the next available exception handler on the stack.

On the Windows platform (the MSVC 2010 target), C++ exceptions are implemented using the native *structured exception handling* (SEH) mechanism. SEH handlers are resident on the stack and form a linked list with a pointer to the head stored in *fs:[0]*. Every C++ function block adds a handler to the head of this list, unless the function has a **`throw()`** or **`noexcept`** declaration. When a C++ exception or a native exception is thrown, the operating system traverses this list, from head to tail, and executes the first handler that is willing to handle the exception. The handler is responsible for unwinding the stack if necessary. If a function with a **`throw()`** declaration throws an exception, there is no handler for the exception for that function block and unwinding does not occur. Because destructors are invoked during stack unwinding, this causes objects to never be destroyed, violating RAII (Resource Acquisition Is Initialization).

Example: Unauthorized Access

This example (see Appendix A) illustrates a situation where compiler hoisting of a **`noexcept`** function out of a *try* block could cause a security violation. Any such case relies on the **`noexcept`** function being nested inside two *try* blocks that catch the same classes of exceptions. This example has the nested *try* block contained in a function that always results in termination of the program.

The overall model is a simple server that handles different types of requests. The server is designed to interact with local and remote clients, and service requests in both cases. Local clients may submit a password with their request, and may then access privileged data. To exploit the hoisted **`noexcept`** function, a remote client connects and requests privileged data. This results in a call to **`client_error()`**, which is intended to always result in program termination. If the request's password length field is too long, a **`bad_alloc`** exception is thrown by **`write_to_security_log()`**, which has an empty **`throw()`** declaration. The design of **`client_error()`** attempts to account for any allocation errors, but because of the **`throw()`** declaration, the **`write_to_security_log()`** function was hoisted out of the **`try`** block. This results in the exception handler for the parent function (**`server()`**) being invoked. In the toy case presented, this results in the server servicing the request despite the server detecting that the client did not have sufficient privileges.

This example illustrates that the use of **`noexcept`** by a programmer without full knowledge of how the compiler will handle it can result in counterintuitive behavior in the resulting binary. This behavior can cause security policy violations ranging from crashes to data leaking. As long as a program relies on the default behavior of **`std::terminate()`** (to abort the program), there are no possible remote code execution exploits enabled by *noexcept*. However, If a program uses a custom *termination function*, **`noexcept`** use may contribute to remote code execution.

Conclusion

We expect that **`noexcept`** will be viewed as a low-risk mechanism for improving performance and that it will be used exhaustively and improperly. To protect users and programmers, violations of **`noexcept`** should not allow a security policy violation. The only option we see as valid is for a program to exit immediately upon violation. The easiest way to implement this option is to call **`std::terminate()`** as is already standard in the case of a **`throw()`** violation.

Appendix A

```
#include "stdafx.h"
#include <stdio.h>
#include <tchar.h>
#include <stdlib.h>
#include <new>
#include <winsock2.h>
#include <windows.h>
#pragma comment(lib,"ws2_32.lib")
using namespace std;
#define BLEN 100

bool is_local_client(unsigned int addr) {
	//Check that addr is within internal network
	if(addr == htonl(0x7f000001))
		return true;
	else
		return false;
}

void send_secure_data(SOCKET client) {
	char* secret = "SECRET DATA\n";
	send(client,secret,strlen(secret),0);
}

void write_to_security_log(char* bad_pass,unsigned int pass_len) throw() {
	char* err_string = new char[pass_len+20];
	//copy pass to string with error info
	//write to log
}

void write_to_log(char* pass) {
	//write info to log about transaction
}

bool check_pass(char* pass) {
	//Check the password somehow
	return false;
}

__declspec(noreturn)  // Terminate always on client errors!
void client_error(char* pass) {
	unsigned int pass_len = ntohl(*(unsigned int*)pass);
	try {
		write_to_security_log(pass+sizeof(unsigned int),pass_len);
		//cleanup
		write_to_log("SERVER: Request terminated");
	}
	catch(bad_alloc) {
		fprintf(stderr,"SERVER: ALLOC ERROR IN CLIENT ERROR\n");
	}
	_exit(10);
}

int server() {
```

```
	WSADATA wsaData;
	WORD version;

	version = MAKEWORD( 2, 0 );
	WSAStartup( version, &wsaData );

	SOCKET server;

	server = socket( AF_INET, SOCK_STREAM, 0 );

	struct sockaddr_in sin;

	memset( &sin, 0, sizeof sin );

	sin.sin_family = AF_INET;
	sin.sin_addr.s_addr = INADDR_ANY;
	sin.sin_port = htons( 4261 );

	if (bind( server, (struct sockaddr*)&sin, sizeof sin ) == SOCKET_ERROR) {
		/* could not start server */
		return FALSE;
	}

	while (listen(server, SOMAXCONN ) == SOCKET_ERROR);

	SOCKET client;
	int length;

	length = sizeof sin;
	client = accept(server, (struct sockaddr*)&sin, &length );

	// Check client connection
	bool local_client = is_local_client(sin.sin_addr.S_un.S_addr);

	try {

	// Get command
	char* pass = new char[BLEN];
	recv(client,pass,BLEN,0);

	//Check if its OK to talk to with this client
	if(!local_client || !check_pass(pass))
		client_error(pass); // function does not return
	}
	catch(bad_alloc) {
		fprintf(stderr,"SERVER: ALLOC ERROR\n");
	}

	//Client has rights, do command
	send_secure_data(client);

	while(1);
	return 0;
}

int main(int argc, char * argv[]) {
	server();
	return 0;
```

}

[i] D. Abrahams, R. Sharoni, D. Gregor, [i]Allowing Move Constructors to Throw N3050, 2010-03-12, http://www.open-std.org/JTC1/SC22/WG21/docs/papers/2010/n3050.html