Microsoft Compiler Flaw Technical Note

Chris Ren, Michael Weber, and Gary McGraw

Abstract

Microsoft is making an important push to improve software security, as evidenced by the Gates memo of January 2002. However, Microsoft clearly has room for improvement if (as we show below) even their security features have architectural security problems.

Microsoft added a new security feature to their latest C++ compiler, called both Visual C++.Net and Visual C++ version 7 that was released February 13th. This security feature is meant to protect potentially vulnerable source code automatically from some forms of buffer overflow attack. The protection afforded by the new feature allows developers to continue to use vulnerable string functions such as strcpy() as usual and still be "protected" against some forms of stack smashing. The new feature is closely based on an invention of Crispin Cowan's called StackGuard and is meant to be used when creating standard native code (not the new .NET intermediate language, referred to as "managed code").

Note that the new feature is meant to protect *any* program compiled with the "protected" compiler feature. In other words, the idea is that using this feature should help developers build more secure software. However, in its current form, the Microsoft feature leads to a false sense of security because it is easily defeated.

Microsoft's feature includes the ability to set a "security error handler" function to be called when a potential attack is underway. Because of the way this was implemented, the Microsoft security feature is itself vulnerable to attack. An attacker can craft a special-purpose attack against a "protected" program, defeating the protection mechanism in a straightforward way.

One elegant feature of StackGuard and its related Microsoft cousin, is the efficiency of the checking mechanisms. However, the mechanism can be bypassed in several ways. The kinds of attack that Cigital made use of to defeat the Microsoft mechanism are neither novel nor do they require exceptional expertise. Had Microsoft studied the literature surrounding StackGuard, they would have been aware of the existence of such attacks.

In our opinion, the StackGuard mechanism makes a poor efficiency/security tradeoff, especially as implemented in Microsoft's compiler. StackGuard is not a perfect approach for stopping buffer overflow attacks. In fact, according to its inventor Crispin Cowan, StackGuard was developed in the context of a fairly serious constraint. Crispin simply patched the gcc code generator so as not to require a new compiler or to re-architect the gcc compiler from the ground up.

There are several well known approaches not based on StackGuard that a compiler-producer might use to defeat buffer overflow attacks. Microsoft chose to adopt a weak solution rather than a more robust solution. This is a design-level flaw leading to a very serious set of potential attacks against code compiled with the new compiler. The Microsoft compiler is thus in some sense a "vulnerability seeder".

Cigital helps companies avoid security problems in software through both education/training (see the book *Building Secure Software* and the training courses based on it) and the application of advanced tools/methods. Instead of relying on a runtime compiler feature to protect against some kinds of string buffer overflows, developers and architects should put in place a rigorous software security regimen that includes source code review. Cigital SecureReview can and should be used to detect potential problems in C++ source code of the sort that the broken Microsoft feature is meant to thwart. Completely removing these problems from code in advance is much better than trying to catch them as they are exploited at runtime.

Bypassing Microsoft's Port of StackGuard

The new /GS compiler option in Visual C++.Net (Visual C++ 7.0) allows developers to build their applications with a so-called 'buffer security check'. In 2001, there were at least two Microsoft-written articles, one by Michael Howard and one by Brandon Bray, published to introduce the option [1,2]. Based on reading the documentation of the /GS option and examining binary instructions generated by the compiler with the option, Cigital researchers have determined that the /GS option is in essence a Win32 port of StackGuard.

Spears and Shields

Overflowing an unchecked stack buffer makes it possible for an attacker to hijack a program's execution path in many different ways. A well known and often used attack pattern involves overwriting the return address on the stack with an attacker's desired address so that a program under attack will jump to the address on function exit. The attacker places attack code at this address which is subsequently executed.

The inventors of StackGuard first proposed the idea of placing a canary before the return address on function entry so that the canary value can be used on function exit to detect whether the return address has been altered. They later improved their implementation by XORing the canary with the return address on function entry to prevent an attacker from overwriting the return address while bypassing the canary [3]. StackGuard turns out to be a reasonable way of preventing some kinds of buffer overflows by detecting them at runtime. A similar tool called StackShield uses a separate stack to store return addresses, yet another way to defeat some kinds of buffer overflows.

Modifying a function return address is not the only way to hijack a program. Other possible attacks that can be used to bypass buffer protection tools like StackGuard and StackShield are discussed in an article in Phrack 56 [4]. Here is the basic idea: if there is a variable of pointer type on the stack *after* a vulnerable buffer, and that variable points somewhere that will be populated with user-supplied data in the function, it is possible to overwrite the variable to carry out an attack. The attacker must first overwrite the pointer variable to make it point to an attacker's desired memory address. Then a value supplied by the attacker could be written to this address. An ideal memory location for an attacker to choose would be a function pointer that will be called later in the program. The Phrack article discussed how to find such a function pointer in the Global Offset Table (GOT). A real world exploit that bypassed StackGuard in this way was published by security focus at URL http://www.securityfocus.com/archive/1/83769.

An Overview of Microsoft's Port of StackGuard

Many details about Microsoft's /GS implementation can be found in three CRT source files, namely seccinit.c, seccook.c, and secfail.c. Others can be found by examining the instructions generated by the compiler with the /GS option.

One 'security cookie' (canary) will be initialized in the call of CRT_INIT. There is a new library call '_set_security_error_handler' which can be used to install a user defined handler. The function pointer to the user handler will be stored in a global variable 'user_handler'. On function exit, the compiler generated instruction 'jmp's to the function '__security_check_cookie' defined in seccook.c, if the security cookie is modified, the '__security_error_handler' defined in secfail.c would be called. The code in '__security_error_handler' first checks whether a user-supplied handler is installed. If so, the user handler will be called. Otherwise, a default 'Buffer Overrun Detected' message is displayed and the program terminates.

There is at least one problem with this implementation. In Windows, something like a writable GOT doesn't exist, so even given the aforementioned layout of the stack, it is not that easy for an attacker to find a function pointer to use. But because of the availability of the variable 'user_handler', an attacker doesn't need to look very far before finding an excellent target!

An Example of Bypassing Microsoft's Port

Let's take a look at the following toy program,

```
#include <stdio.h>
#include <string.h>

/*
        request_data, in parameter which contains user supplied encoded string like
                "host=dot.net&id=user_id&pw=user_password&cookie=da".
        user_id, out parameter which is used to copy decoded 'user_id'.
        password, out parameter which is used to copy decoded 'password'
*/
void decode(char *request_data, char *user_id, char *password){
        char temp_request[64];
        char *p_str;

        strcpy(temp_request, request_data);
        p_str = strtok(temp_request, "&");
```

```
        while(p_str != NULL){
                if (strncmp(p_str, "id=", 3) == 0){
                        strcpy(user_id, p_str + 3 );
                }
                else if (strncmp(p_str, "pw=", 3) == 0){
                        strcpy(password, p_str + 3);
                }
               p_str = strtok(NULL, "&");
        }
}

/*
        Any combination will fail.
*/
int check_password(char *id, char *password){
        return -1;
}
/*
        We use argv[1] to provide request string.
*/
int main(int argc, char ** argv)
{
        char user_id[32];
        char password[32];

        user_id[0]  = '\0';
        password[0] = '\0';

        if ( argc < 2 ) {
                printf("Usage: victim request.\n");
                return 0;
        }

        decode( argv[1], user_id, password);

        if ( check_password(user_id, password) > 0 ){
                //Dead code.
                printf("Welcome!\n");
        }
        else{
                printf("Invalid password, user:%s password:%s.\n", user_id, password);
        }

        return 0;
}
```

The function ‘decode’ contains an unchecked buffer ‘temp_request’, and its parameter ‘user_id’ and ‘password’ can be overwritten by overflowing ‘temp_request’.

If the program is compiled with the /GS option, it is not possible to alter the program’s execution path by overflowing the return address of the function ‘decode’. But it *is* possible to overflow the parameter ‘user_id’ of the function ‘decode’ to make it point to the aforementioned variable ‘user_handler’ first! So when ‘strcpy(user_id, p_str + 3);’ is called, we can assign a desired value to ‘user_handler’. For example, we can make it point to the memory location of ‘printf("Welcome!\n");’, so that when the buffer overflow is detected, there would appear to be a user installed security handler and the program will execute “printf("Welcome!\n");”. Our exploit string looks like this:

id=[location to jump to]&pw=[any]AAAAAAA…AAA[address of user_handler]

Given a compiled “protected” binary, determining the memory address of ‘user_handler’ is trivial given some knowledge of reverse-engineering. The upshot is that a protected program is actually vulnerable to the kind of attack it is supposedly protected from. Also worth mentioning is the fact that the canary itself can be hijacked in a closely related attack.

Solutions

There are several alternative paths that can be followed to fix this problem. The best solution involves having developers adopt a type safe language such as Java. The next best solution is to compile in dynamic checks on string functions that occur at runtime (though the performance hit must be accounted for). These solutions do not always make sense given project constraints.

Modifying the current /GS approach is also possible. The main goal of each of the suggested fixes below is to achieve a higher level of data integrity.

- Ensure the integrity of stack variables by checking the canary more aggressively. If a variable is placed *after* a buffer on the stack, a sanity check should be performed before that variable is used. The frequency of such checks can be controlled by applying data dependence analysis.

- Ensure the integrity of stack variables by rearranging the layout of the stack. Whenever possible, local non-buffer variables should be placed *before* buffer variables. Furthermore, since the parameters of a function will be located after local buffers (if there are any), they should be treated as well. On function entry, extra stack space can be reserved before local buffers so that all parameters can be copied over. Each use of a parameter inside the function body is then replaced with its newly created copy.

Work on this solution has already been done by at least one IBM research project [5].

- Ensure the integrity of global variables by providing a managed-writable mechanism. Very often, critical global variables become corrupted due to program errors and/or intentional abuse. A managed-writable mechanism can place a group of such variables in a read-only region. When modifying a variable in the region is necessary, the memory access permission of the region can be changed to 'writable'. After any such modification is done, its permission is changed back to 'read-only'. With such a mechanism, an unexpected 'write' to a protected variable results in memory access violation. For the kind of variable that only gets assigned once or twice in the life of a process, the overhead of applying a managed-writable mechanism is negligible.

Future work

There are several avenues open for future work surrounding the Microsoft technology and related approaches. Research should be performed to examine how random the canary initialization algorithm can and should be. If an application is designed to run as a service on a host and an attacker is able to figure out the boot time of the host, the effectiveness of the buffer overrun protection could be compromised if the application's designer chooses to install their own security error handler so that a thread can be terminated instead of terminating a process upon detecting a buffer overflow.

Work at Cigital on weaving security into existing source code using the Aspect Oriented Programming paradigm is also directly relevant.

References

[1] New Visual C++.NET Option Tightens Buffer Security
http://security.devx.com/bestdefense/2001/mh0301/mh0301-1.asp
[2] How Visual C++ .NET Can Prevent Buffer Overruns
http://www.codeproject.com/tips/gsoption.asp
[3] StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks http://www.immunix.org/documentation.html
[4] Bypassing Stackguard And Stackshield http://www.phrack.org/show.php?p=56&a=5
[5] GCC Extension For Protecting Applications From Stack-Smashing Attacks
http://www.trl.ibm.com/projects/security/ssp/