SysGroup

Buffer Overflows and How They Are Exploited

A White Paper

by Gavin Solomon

16 February 2026

TRUSTSTREAM CYBERSECURITY

A Subsidiary of SysGroup Plc

Table of Contents

Abstract

One of the most publicized types of vulnerability, casually spoken about yet poorly understood by both defensive security and offensive security practitioners, is known as the "buffer overflow". On the one hand, regular security bulletins draw frequent attention to the latest security updates of application and operating systems, warning of a previously undiscovered "buffer overflow". On the other hand, whenever penetration testers identify software or OS versions with a known "buffer overflow" vulnerability, they immediately search for a published exploit. Their aim is to demonstrate, with as little effort as possible, how a system or application can be compromised.

In other words, talk of a "buffer overflow" immediately attracts the attention of both defensive teams and penetration testers alike. In the one case, it is because it is known to be a flaw with potentially serious security consequences. In the other case, it is seen as a golden hacking opportunity. Yet few practitioners understand genuinely what a buffer overflow really is and why it represents such a serious danger. Defensive teams merely provide generic advice to IT professionals: "patch your systems" and refer them to the advisory of the relevant vendor. Penetration testers and hackers usually only act like "script kiddies": they find a published exploit for the software version they are exploiting, and check that the description matches the words "buffer overflow".

This paper aims to explain what a buffer overflow is, how previously unknown buffer overflow vulnerabilities are detected and how they can be abused by exploit developers to compromise vulnerable software. The paper will also demonstrate the aforementioned with a practical example, using the TTPs of malware developers, but will stop short of developing a complete sample exploit. The aim is not to encourage the development of malware, but to bring to life the principles of how buffer overflows are abused with an actual example.

Tools Used in the Demonstrations

The demonstrations shown in this paper follow similar steps to the method described by Offensive Security (2020) in their publication Penetration Testing with Kali Linux. The tools used, however, are significantly different as is the target application.

The app was developed and run on the following platform: Windows 10 Home 22H2
App was developed with: Visual Studio 2026
Visual Studio components installed:

- Desktop development with C++
- ASP.NET and web development
- Python development
- .NET desktop development

x64dbg, packaged with x32dbg
checksec and xAnalyzer plugins for x64dbg
Python 3.13.12

The source code for the vulnerable app, boverflow1, and for the vulnerable DLL that it calls, vulnlib.dll, is given in the appendices. The DLL, once generated, should be placed in the same folder as the executable, boverflow1.exe.

As this is only a demonstration of the principle of exploiting buffer overflows, the app being developed will be entirely a command-line app. Exploitation will take place from the command line on the same system.

Introduction: What is a Buffer Overflow?

Put simply, a buffer is an area of memory reserved for holding data temporarily. Most often, it will contain a string or a sequence of printable characters, although this does not have to be. Typically, it will be declared as an array as in the following C statement:

char name[10];

This tells the compiler to reserve an area of memory large enough to contain 10 characters and to treat it as a variable called “name”. Each character in the name can be referenced individually, where name[0] is the first character and name[9] the last.

A buffer overflow is what happens when the number of characters entered is more than the buffer has room to store and when the remaining characters overwrite other reserved areas of memory. This happens when no checks are performed on the amount of space available.

So, what happens when data is written that goes beyond the end of the buffer? Does it overflow into the next buffer declared, as in the following code?

char buffer1[10];
char name[10];
char buffer2[10];

If we attempt to write the word “abracadabra” into buffer name, does the final “a”, which is the 10th letter, become stored at the start of buffer2?

In fact, this is not the case. Instead, it actually is stored in the previously declared buffer, buffer1. This is because variables are stored in an area of memory called the “stack”. On an Intel architecture, the stack grows downwards, not upwards. So name is stored in a lower area of memory than where buffer1 is held.

The following section will demonstrate how writing too many characters to name will result in the previously declared variable being affected.

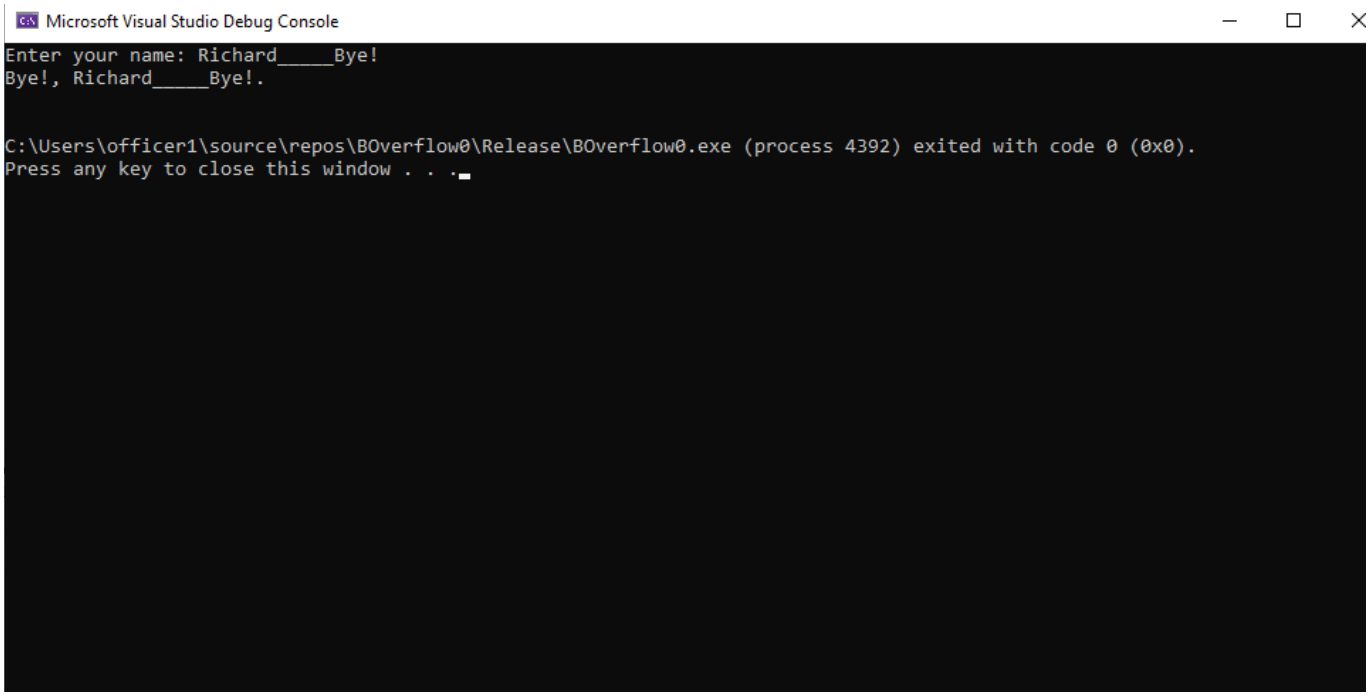
A Simple Demonstration of a Buffer Overflow

We shall start by demonstrating the above situation. The above code has been incorporated into a short program that asks the user for their name and reprints it with a greeting. The name variable has been declared to hold 10 characters.

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

int main(void)
{
        char greeting[8];
        char name[10];
        printf("This is subroutine1.\n\n");
        strcpy(greeting, "Hello\0"); // Initialization
        printf("Enter your name: ");
        scanf("%s", name);
        printf("%s, %s.\n\n", greeting, name);
        return 0;
}
```

The following screenshot shows the normal operation of the program.

The next screenshot shows the operation of the program when the name overspills the number of characters made available for it. While the name has only 10 characters reserved for it, going beyond the 10 characters will not in practice result in an immediate buffer overflow. This is because the Visual C compiler will pad the “name” buffer with extra bytes, so that the resulting space available will be a multiple of 16. If we enter in excess of 16 characters, however, the name will overspill into the greeting as shown below:



In the above demonstration, the characters entered into the name include underscores followed by the word “Bye!”. These characters have been calculated to overspill into the greeting. The result is that the greeting “Hello” has become “Bye!”.

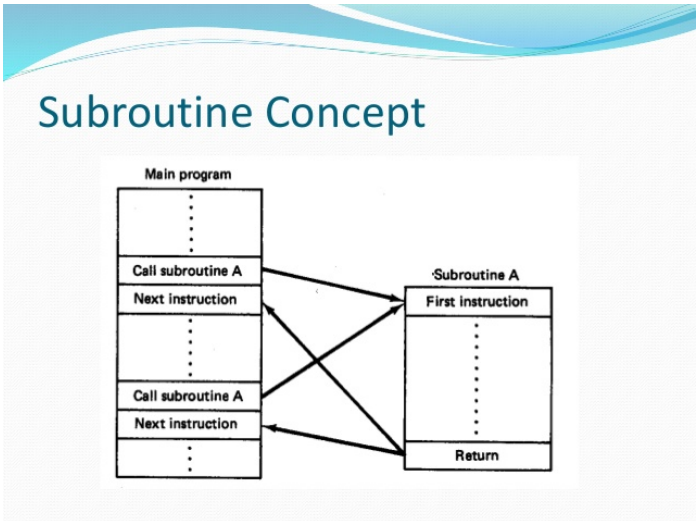
At first site it might appear that the greeting has been displayed twice, once before the name and once afterwards. In fact, this is not the case. The second appearance of the word “Bye!” is actually part of the name being printed. As it overspills the machine does not know where to stop. The output continues until it meets a null character (hex code 0x00), which terminates the string. This could be anywhere.

The Dangers of a Buffer Overflow

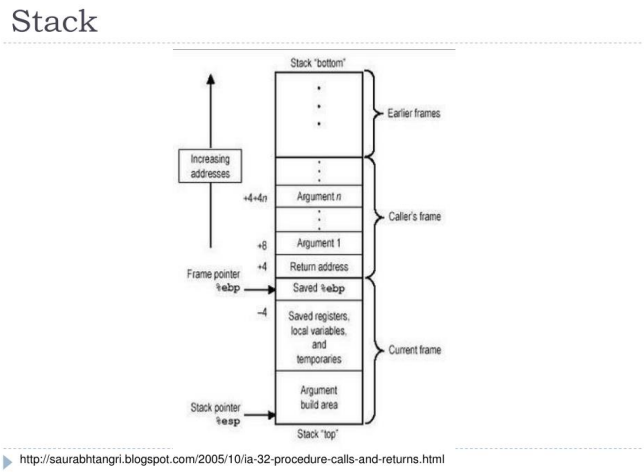
The above demonstration may seem harmless enough. When invited to enter a name, we entered too many characters. The result was that the greeting “Hello” was overwritten with something else of our choice. But is this dangerous?

The obvious answer is, yes it is, because there are many circumstances where changing the data being processed, especially if the user was not meant to change it, can have unintended and even costly consequences. Beyond this, though, there is an even greater yet less obvious danger. Continuing to write far enough beyond the end of the buffer can cause a computer to “forget” where it was within the instructions it was carrying out and change the execution flow. The result of a buffer overflow, therefore, could be that the execution is directed to some random address that has overwritten the instruction address that the machine was supposed to go to. The reasons why this happens is explained by a part of memory known as the “stack”.

Almost every machine-executable app/program is made up of subroutines, each of which carries out a subtask. Even if you only have one set of main instructions, the compiler usually encapsulates this as a subroutine of another main task. But when a subroutine is called, the machine needs to remember where to return after it completes.



The stack is an area of memory, where the last item to be added is also the first item to be retrieved (last-in-first-out). One of its most important functions is to remember the memory address where the CPU must return when the current subroutine finished. Each time a is called subroutine, maybe within another subroutine, the return address is added to the stack. This is also the first place where it returns.



The illustration above shows how the stack grows in an Intel architecture. The stack actually grows downwards in memory. So, the “top” of the stack has the lowest memory address.

In the current subroutine, the local variables are stored at the top of the stack, i.e. in the lower memory areas. If a local buffer is overwritten, initially it will overwrite other variables and saved registered. But if the overspill goes far enough, it will overwrite the return address. This will cause the machine, on finishing the subroutine, to continue to read instructions and execute instructions held at a different address from what was intended.

If the return address was overwritten by accident, then at worst this will probably result in a computer crash as random data is interpreted as instructions. But if the execution flow can be changed predictably, by overwriting the return address with an address containing instructions that an attacker would like executed instead of the intended instructions, then the result is that the CPU could fall under the control of the attacker.

A Simple Demonstratoin of The Danger

To demonstrate how the execution flow can be changed, we shall take the same program as before but enter a considerably longer piece of text in the name. On this occasion, however, we shall get the program to print extra diagnostic messages about the stack, where the most recent stack element is located and what return address is stored. Below is a modified version of the program:

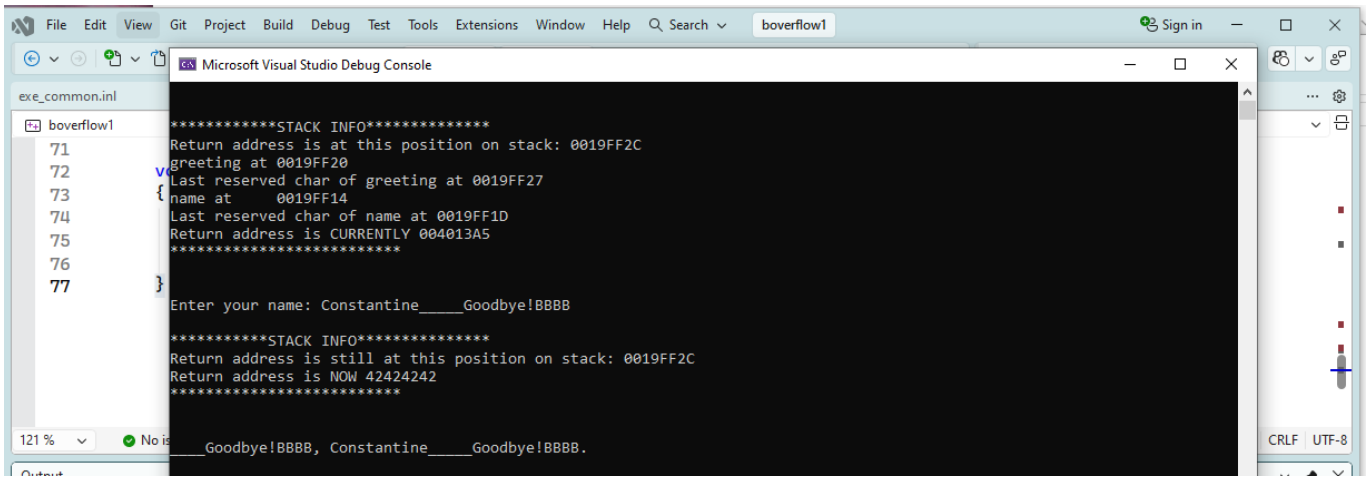
```
#define _CRT_SECURE_NO_WARNINGS
#include <stdio.h>
#include <string.h>
#include <intrin.h>
#include <windows.h>

void subroutine1(void);
void subroutine2(void);

typedef int (*GetSysInfoFn)(char*, size_t, char*, size_t);
typedef int (*DummyFunc)(void);

int main(void)
{
	char buffer1[128];
	char buffer2[128];
	int retCode;

	HMODULE h = LoadLibraryA("vulnlib.dll"); // Load DLL from same folder

	GetSysInfoFn GetSysInfo = (GetSysInfoFn)GetProcAddress(h, "platformInfo");
	DummyFunc myFunc = (DummyFunc)GetProcAddress(h, "dummyFunc");

	printf("About to call subroutines.\n");

	retCode = GetSysInfo(buffer1, sizeof(buffer1), buffer2, sizeof(buffer2));
	subroutine1();
	subroutine2();

	return 0;
}

void subroutine1(void)
{
	char greeting[8];
	char name[10];
```

```
    printf("This is subroutine1.\n\n");

    strcpy(greeting, "Hello\0"); // Initialization

    /* DEBUG INFO */
    printf("\n**********STACK INFO**********\n");
    void* rsp = _AddressOfReturnAddress(); // Points into the current stack frame
    printf("Return address is at this position on stack: %p\n", rsp);
    printf("greeting at %p\n", (void*)greeting);
    printf("Last reserved char of greeting at %p\n", (void*)&greeting[8]);
    printf("name at     %p\n", (void*)name);
    printf("Last reserved char of name at %p\n", (void*)&name[10]);
    void* rip = _ReturnAddress();
    printf("Return address is CURRENTLY %p\n", (void*)rip);
    printf("**************************\n\n\n");
    /* END OF DEBUG */

    printf("Enter your name: ");
    scanf("%s", name);

    /* DEBUG INFO */
    printf("\n**********STACK INFO**********\n");
    rsp = _AddressOfReturnAddress(); // Points into the current stack frame
    printf("Return address is still at this position on stack: %p\n", rsp);
    rip = _ReturnAddress();
    printf("Return address is NOW %p\n", (void*)rip);
    printf("**************************\n\n\n");
    /* END OF DEBUG */

    printf("%s, %s.\n\n", greeting, name);


void subroutine2(void)
{
    printf("**************************\n");
    printf("This is subroutine2.\n\n");
    printf("**************************\n");
}
```

If we now enter more than 56[1] characters of data, as described in the previous section, then we shall overwrite part, if not all, of the return address. The screenshot below shows

[1] The number of characters or bytes may not be 56. It depends on whether you are running in debug mode and on the different compiler options. The compiler often organizes the memory management to pad the reserved areas with extra bytes for various reasons, e.g. to achieve alignment or insert “canary values”.

this very scenario. The program now prints diagnostic messages about the memory address of name, greeting and the return address held on the stack. You will notice that the return address held on the stack has changed as a result of the user's entering too much data:



Note that the return address now contains the hexadecimal characters 42 42 42 42 42..... In decimal 0x42 is 66, which is the character code for to the letter "B", which we entered multiple times when we typed the user's name.

As predicted, the program subsequently crashed with a negative exit code rather than exiting gracefully.

How Exploit Developers Abuse Buffer Overflows

So far, we have demonstrated how a buffer overflow might be used to change data that should not be alterable by a user, or to crash a program. A program crash could be caused either accidentally by a user who enters too much data, or by design where an attacker is trying to mount a denial-of-service attack.

Yet the most valuable achievement from an attacker's perspective is not simply a denial of service. This may cause costly disruption, but draws attention because it is immediately noticed. What an attacker values more is the ability to gain control of a system. A denial of service can always then be carried out at a later stage.

The previous demonstration showed how an attacker can cause a system to crash by overwriting the return address with junk, in this case ("BBBB"). But to gain control of a system, the attacker must not simply overwrite the return address with junk or random data. They must overwrite it so that the CPU is redirected to code that they control. This code could be stored, as op codes, somewhere in the data they have supplied.

This might sound simple. You might think that the attacker simply has to overwrite the buffer far enough to reach the return address, as we have done previously, and then add further data containing malicious code. They then have to make sure that the return address points the CPU to the malicious code that follows.

In fact, doing this directly is laden with difficulties. It is almost impossible to set the return address directly to the start of the malicious code. This is because, each time the program is run, it is loaded into a different space in memory. Only the internally layout of the program's address space Is predictable (unless ASLR is in effect – see section on limitations). Therefore, the attacker cannot predict reliably where their malicious code is going to start. An exploit is unlikely to work if it always points to the same place, unless we can predict what code will be there.

So that we do not have to predict exactly where the stack will be loaded in memory, we have to leverage an instruction that is already in the legitimate, existing, code. Ideally, this will be an instruction that uses relative addressing or a similar mechanism that can redirect the execution flow to where we want, regardless of the base address where the app is loaded. The next section will demonstrate how this is done.

Step	Method
Determine whether the app has a buffer-overflow vulnerability.	Supply increasingly long user-input data to the different fields where user input is accepted. Continue until the program crashes.
Find out the position where the return address starts in the overflowing buffer.	Find the exact buffer length where just adding one more character causes the program to crash. This extra character is the first byte of the return address held on the stack.
Find all the bad characters, i.e. byte values that will either be stripped out by the app or cause the app to stop reading user input.	Create a Python script that supplies user input containing every hex value from 0x00 to 0xFF in sequence. If none of the data is accepted, remove the first character (0x00) and rerun the script. If the input is accepted, but stops after only a few characters, remove the first character that is not accepted. Continue re-running removing characters that are not read until all the bytes are accepted. Note the characters that were not accepted.
Find a static library that the app uses.	Load the app in a system-level debugger such as Immunity or x32dbg with a plugin like mona.py (for Immunity) or checksec (for x32dbg). Run the app until the point where user input is accepted. List the modules loaded and find one that loads at a static address without certain security checks (see next section).
Find a JMP ESP instruction in the existing app or in one of the libraries that it loads.	Do a memory search for the bytes 0xFFE4. Check that the memory address of the instruction does not contain any bad byte values.
Create malicious shellcode.	Use a tool to create a payload and specify which bad byte values to avoid.
Add the Shellcode to your Python script.	The payload will typically be stored as a Byte array or binary object. This should be appended to the end of the buffer supplied by your Python script. Because of alignment and other issues, it may have to be preceded by several NOP (no operation) bytes.

Demonstration – Developing an Exploit

One of the most useful machine-level instructions for redirecting execution predictably to an area that contains the code we want is the JMP ESP instruction. This instruction redirects execution to the address indicated by the stack pointer:

	Address	Stack Contents (overwritten)
		.....malicious code...
		.....malicious code...
stack pointer after return --→		.....malicious code
		return address
		saved stack base pointer (EBP)
		greeting (padded to 32 bytes)
current stack pointer --→		name (padded to 16 bytes)
		...next available space...

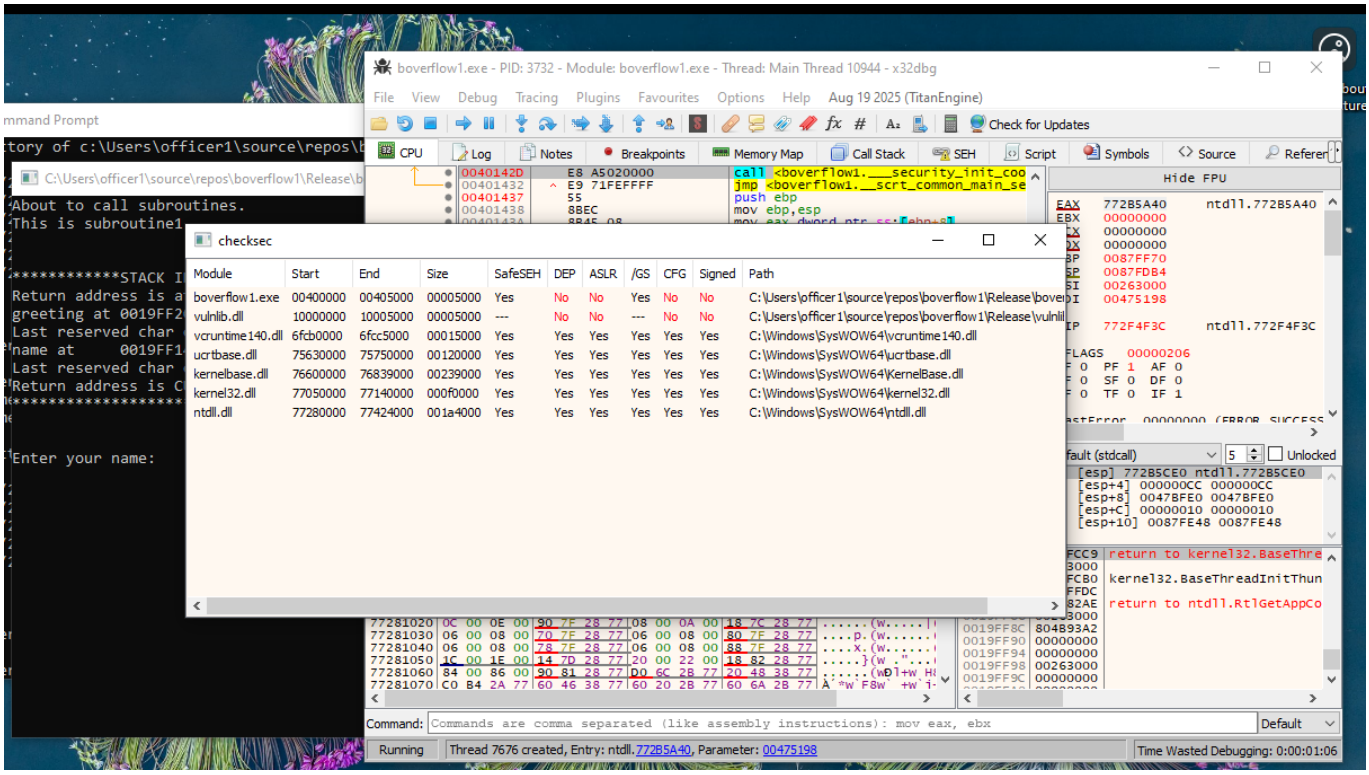
The "name" buffer will be overwritten so that not only is the greeting changed, but also the items shown further upwards in the stack, including the return address. Above the return address we shall have the malicious code stored. The trick is to overwrite in the return address with the address of an existing JMP ESP instruction. Once the CPU encounters a "RET" instruction, it will "return" to this address instead of the original one that was in its place. After the RET instruction the stack pointer will be pointing to the next slot above the return address, where the malicious code begins, but the CPU will have been directed to a JMP ESP instruction. After executing JMP ESP, the CPU will go to the start of the malicious code, where the stack pointer is pointing.

We now need to find an existing JMP ESP instruction. To do this we load the program and see which modules are loaded together with it. This can be done using the *checksec* plugin. The screenshot below shows the result of the *checksec* analysis when the program has reached the statement asking for the user's name:

Modules Loaded (Analysis by Checksec Plugin)

We now need to look at the loaded modules to find one where the address loaded is fixed. The screenshot below shows boverflow1.exe loaded in the x32dbg debugger with the checksec plugin. The plugin shows all the where the boverflow1.exe app is loaded and also

lists other shared libraries (DLLs – dynamic link libraries) used by the app. These DLLs contain subroutines that are stored separately, but used by this app and others.

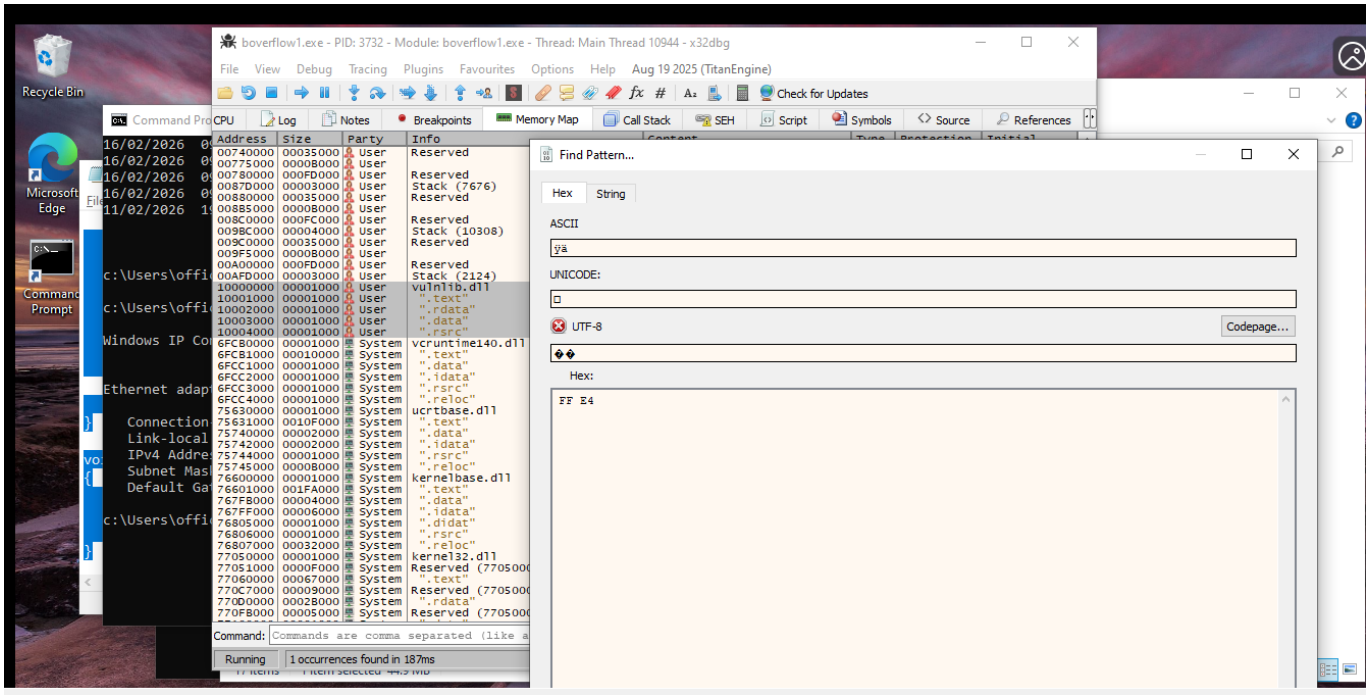


Before we look for a JMP ESP instruction in the existing code, we need to find a suitable library that we can search. From the flags in this output, we can see that the vulnlib.dll library has SafeSEH, (Structured Exception Handler Overwrite, an exploit-preventative memory protection technique), ASLR, and NXCompat (DEP protection) disabled. As Offensive Security Ltd (2020: 391) explain when demonstrating a similar vulnerability with another vulnerable app: “In other words”, it “has not been compiled with any memory protection schemes” and “will always reliably load at the same address, making it ideal for our purposes.”

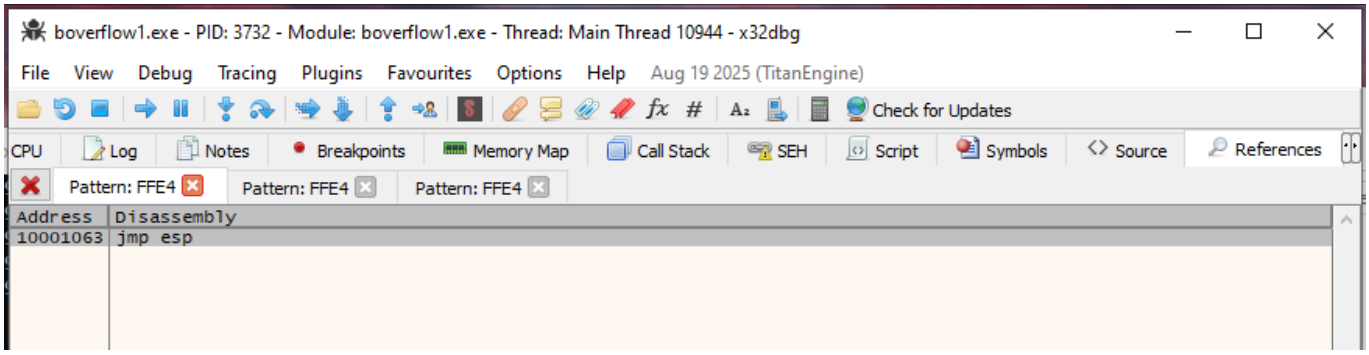
We now need to search this library to look for a JMP ESP instruction that we can leverage. Once we know the address of the JMP ESP instruction, we can write that address to the input stream in such a position that it will overwrite the return address. Instead of returning from the subroutine to where it came from, the CPU will branch to the JMP ESP instruction. When this is executed, the CPU will jump to the current stack pointer, which will be the first memory address above the return address. If we have calculated everything correctly, this will contain our malicious code.

Since we know from the above screenshot that vulnlib.dll is loaded at starting address 0x10000000 and finishes at address 0x10005000, we must search that area of memory for a JMP ESP instruction. The screenshot below shows the memory map in x32dbg. Here we

have highlighted the memory range where vulnlib.dll is loaded. We now search for the hex bytes 0xFF 0xE4. These are the opcodes for "JMP ESP".



In the screenshot below, we can see that the find command has returned the address of the bytes 0xFF 0xE4, together with a disassembly confirming that the operation is indeed a JMP ESP.



Ideally, therefore, we should add extend the stream of input data to overflow the buffer and overwrite the return address with 0x10001063. As we shall see later, there is a problem with the bytes in this address, but below is the principle:

Byte Position	0-15	16-47	48-55	56-63	64.....
Content	Richard...	Bye!.....	...garbage...	10 00 10 63	...malicious shellcode..

Unfortunately, with this program we would never be able to overwrite the return address with 0x10001063, because the byte 0x00 cannot be supplied as part of the user input. When using the standard input string, 0x00 is known as a null character and it is used to indicate the end of a piece of text. When the machine encounters 0x00, it stops reading any further input. 0x00 is one example of what an exploit developer calls a “bad character”. At this point we therefore have to search for a different memory address that contains a JMP ESP instruction, or alternatively find another branching instruction that redirects execution to an address that can be predicted regardless of where the program loads in memory.

0x00 is not likely to be the only bad character. In the case of user text input, the characters 0x0A (line feed) and 0x0D (carriage return) are also likely to cause the machine to stop reading the input. Before the developer can find a suitable branching instruction, they must first find all the bad characters, i.e. all the character codes that cannot reliably be supplied to the program they are exploiting. There are techniques for achieving this, but this is beyond the scope of this paper.

Not only must bad bytes be avoided in the return address, but the malicious shell code that the developer adds to their input must also avoid these bad bytes, because any shell code supplied has to form part of the user-supplied input. Tools used to create shellcode disguised as data within a Python script optionally allow you to specify which byte values to avoid. As the purpose of this paper is to develop an understanding of how buffer overflows are exploited, rather than provide an instruction manual on how to carry out the exploit, this is out of scope.

Limitations

Anyone who has followed the demonstration will question the usefulness, from the point of view of the hacking community, of exploiting a command-line app locally, from the same machine as the one on which it is running. After all, you would already need local access to the machine to run the app in the first place, meaning that either you already have credentials, or the machine has no authentication. In addition, you could justifiably argue that the app is only running in the security context of the logged-on user. Therefore, even if you successfully manipulate the app into running code that was not intended, you could simply have run the same code directly yourself with the same level of privileges.

In fact, this is not entirely true in all contexts. In Linux, for example, a program can be set to run with superuser privileges, regardless of the user who runs it. For some utility programs, such as ping, this is necessary. If such a program can be manipulated to run arbitrary code as a superuser, the possibilities for a local attacker with limited privileges are greatly enhanced.

More importantly, whether a program is a local app that accepts input over the command line, or a service that accepts input across a network, makes little difference to the principle of how a buffer overflow is exploited. A user accessing the app as a network service is not logged on to the hosting machine, but the app is always running under a user account. Once the app can be manipulated into executing arbitrary code, an attacker can launch any set of instructions that the app's user account is permitted to carry out. This is particularly dangerous if the app is running under an administrative or root account. For this reason, administrators are always advised not to run network services under administrative accounts.

You will also note that the vulnerable app that we have exploited was a 32-bit app. This was necessary to simplify the exploitation, because the JMP ESP instruction, on which redirecting the execution flow heavily depended, can only be carried out in 32-bit mode. Yet even though the JMP ESP instruction is not available in 64-bit mode, 64-bit Windows applications are not immune to exploitation of buffer overflows. There are other ways in which control-flow structures can be corrupted predictably in Windows 64-bit applications that have been carelessly designed.

The most significant limitation with the demonstration, perhaps, is that modern systems have automatic safeguards against buffer overflow vulnerabilities and their exploitation. These safeguards are enabled by default and work at multiple levels, both at the OS level and at the development level where a programmer tries even to compile source code that is likely to be vulnerable. If you were to try to repeat the steps of the above demonstrations, they would fail unless you carried them out in a specially adapted environment. Below is an outline of the environment you would need to create to reproduce the results of the demonstration above.

The vulnerable application has been run on a platform that was deliberately insecure. For example, the underlying OS was **Windows 10 22H2 (10.0.19045)**, which has been out of support for roughly 4 months at the time of writing. In addition, the following key security safeguards had been intentionally disabled in the Windows settings for the purposes of demonstration:

- Control Flow Guard (CFG)
- Data Execution Prevention (DEP)
- Address Space Loading Randomization (ASLR)
- Randomize Memory Allocations (Bottom-Up ASLR)
- Validate Exception Chains (SEHOP)
- Validate Heap Integrity

Even when the above security safeguards are disabled at OS level, the default options in Microsoft Visual Studio, which was used to develop the vulnerable app, add checks to the application that still perform DEP and ASLR. These checks are built into the app at compile time or at link time. Over and above the previously mentioned safeguards, the compiler by default adds checks specifically for stack buffer overflows. The following project settings were used inside Visual Studio for building the app and the DLL:

- compiler stack security checks were disabled (/GS-)
- the default C-language standard was used by the compiler (legacy MSVC)
- compiler conformance mode was set to No (/permissive)
- the compiler option Enable Address Sanitizer was set to No.
- the linker option Data Execution Prevention was set to No (/NXCOMPAT:NO)
- the linker option Randomized Base Address was set to No (/DYNAMICBASE:NO)
- the linker option Image has Safe Exception Handler was set to No (/SAFESEH:NO)
- code analysis was disabled.

The target architecture for the projects, i.e. both the vulnerable application and the associated DLL, was set to Win32. Note that, although WoW64 allows 32-bit applications to run on 64-bit Windows, it does not allow 64-bit applications to call 32-bit DLLs. 32-bit DLLs must be called only by 32-bit applications, while 64-bit applications must call 64-bit DLLs if they use any DLLs whatsoever.

Results / Conclusion

The exploit of any vulnerability has one of three aims: denial of service (crashing the system or a running service), a foothold on the system with the ability to run commands or privilege escalation. Where a buffer overflow vulnerability exists on a local system, where a user can already run commands, an exploit aims to abuse the buffer overflow in order to execute commands with additional privileges. Where a buffer overflow exists on a network service, an exploit aims to abuse it to carry out either remote command execution (RCE) or a denial of service. If the service is running under a privileged account, RCE can be used to elevate privileges by calling administrative functions as well as execute other commands.

While buffer overflows may enable RCE or privilege escalation, a buffer overflow is not synonymous with these latter terms. A buffer overflow is just one vulnerability that can lead to RCE or privilege escalation, but it is not the only one. We hope that this paper has given its readership some understanding of what a buffer overflow really is and the connection between the vulnerability and RCE and privilege escalation.

We have also sought to explain how exploit developers leverage such a vulnerability, but stopped short of writing a full instruction manual on exploit development. We have demonstrated how buffer overflows are abused on 32-bit Windows systems, but the same principles apply with differences in architecture to 64-bit Windows systems, Linux, BIOS/UEFI firmware and operational technology (OT). The latter, in particular, is a cause for concern, as control systems in industrial environments were often designed long before the growth of the Internet, when network security was of little concern. It is likely that many OT systems are still highly vulnerable to buffer overflows.

Appendices

1. Sources/References

Offensive Security (2020), Penetration Testing with Kali Linux.

Sanchhaya Education Private Ltd (2025), *GeeksForGeeks*, 'Subroutine, Subroutine Nesting and Stack Memory', <https://www.geeksforgeeks.org/computer-organization-architecture/subroutine-subroutine-nesting-and-stack-memory/> [accessed 17 February 2026].

Tangri, Saurabh (2005), *IA-32 Procedure Calls and Returns*, <https://saurabhtangri.blogspot.com/2005/10/ia-32-procedure-calls-and-returns.html> [accessed 17 February 2026].

Zohra, Syeda (2020), *Subroutines*, *Blogspot.com*, <https://zoman1.blogspot.com/2020/01/subroutines.html> [accessed 17 February 2026].

2. Source Code Listings

boverflow1 – main.cpp (single source file, console application)

```
#define _CRT_SECURE_NO_WARNINGS

#include <stdio.h>
#include <string.h>
#include <intrin.h>
#include <windows.h>

void subroutine1(void);

void subroutine2(void);

typedef int (*GetSysInfoFn)(char*, size_t, char*, size_t);
typedef int (*DummyFunc)(void);

int main(void)
{
        char buffer1[128];
        char buffer2[128];
        int retCode;

        HMODULE h = LoadLibraryA("vulnlib.dll"); // Load DLL from same folder
```

```
	GetSysInfoFn GetSysInfo = (GetSysInfoFn)GetProcAddress(h, "platformInfo");
	DummyFunc myFunc = (DummyFunc)GetProcAddress(h, "dummyFunc");

	printf("About to call subroutines.\n");

	retCode = GetSysInfo(buffer1, sizeof(buffer1), buffer2, sizeof(buffer2));
	subroutine1();
	subroutine2();

	return 0;
}

void subroutine1(void)
{
	char greeting[8];
	char name[10];

	printf("This is subroutine1.\n\n");

	strcpy(greeting, "Hello\0"); // Initialization

	/* DEBUG INFO */
	printf("\n**********STACK INFO**********\n");
	void* rsp = _AddressOfReturnAddress(); // Points into the current stack frame
	printf("Return address is at this position on stack: %p\n", rsp);
	printf("greeting at %p\n", (void*)greeting);
	printf("Last reserved char of greeting at %p\n", (void*)&greeting[sizeof(greeting)-1]);
	printf("name at     %p\n", (void*)name);
	printf("Last reserved char of name at %p\n", (void*)&name[sizeof(name)-1]);
	void* rip = _ReturnAddress();
	printf("Return address is CURRENTLY %p\n", (void*)rip);
	printf("**************************\n\n\n");
	/* END OF DEBUG */

	printf("Enter your name: ");
	scanf("%s", name);

	/* DEBUG INFO */
	printf("\n**********STACK INFO**********\n");
	rsp = _AddressOfReturnAddress(); // Points into the current stack frame
	printf("Return address is still at this position on stack: %p\n", rsp);
	rip = _ReturnAddress();
	printf("Return address is NOW %p\n", (void*)rip);
	printf("**************************\n\n\n");
	/* END OF DEBUG */
```

```
        printf("%s, %s.\n\n", greeting, name);
}

void subroutine2(void)
{
        printf("**************************\n");
        printf("This is subroutine2.\n\n");
        printf("**************************\n");
}
```

vulnlib.dll

This was a single user-created source file, dllmain.cpp. All other source and header filed were generated automatically by Visual Studio when a blank DLL was selected as a template.

```
// dllmain.cpp : Defines the entry point for the DLL application.
#include "pch.h"
#include <windows.h>
#include <stdio.h>

typedef LONG(WINAPI* RtlGetVersionPtr)(PRTL_OSVERSIONINFOW);

extern "C" __declspec(dllexport) int dummyFunc();
extern "C" __declspec(dllexport) int platformInfo(char* cpuBuf, size_t cpuBufSize, char* osBuf,
size_t osBufSiz);

BOOL APIENTRY DllMain( HMODULE hModule,
                       DWORD  ul_reason_for_call,
                       LPVOID lpReserved
                     )
{
  switch (ul_reason_for_call)
  {
  case DLL_PROCESS_ATTACH:
  case DLL_THREAD_ATTACH:
  case DLL_THREAD_DETACH:
  case DLL_PROCESS_DETACH:
    break;
  }
  return TRUE;
}
```

```
extern "C" __declspec(dllexport) int dummyFunc()
{
  __asm
  {
    jmp esp;
  }

  return 0;
}

extern "C" __declspec(dllexport) int platformInfo(
  char* cpuBuf, size_t cpuBufSize,
  char* osBuf, size_t osBufSize)
{
  if (!cpuBuf || !osBuf || cpuBufSize == 0 || osBufSize == 0)
    return 1; // invalid parameters

  SYSTEM_INFO si;
  GetNativeSystemInfo(&si);

  const char* cpuStr = "Unknown CPU";

  switch (si.wProcessorArchitecture) {
  case PROCESSOR_ARCHITECTURE_AMD64:
    cpuStr = "x64 (AMD64)";
    break;
  case PROCESSOR_ARCHITECTURE_INTEL:
    cpuStr = "x86 (32-bit)";
    break;
  case PROCESSOR_ARCHITECTURE_ARM:
    cpuStr = "ARM";
    break;
  case PROCESSOR_ARCHITECTURE_ARM64:
    cpuStr = "ARM64";
    break;
  default:
    cpuStr = "Unknown architecture";
    break;
  }

  // Copy CPU string safely
  if (strlen(cpuStr) + 1 > cpuBufSize)
    return 2; // insufficient buffer
  strcpy_s(cpuBuf, cpuBufSize, cpuStr);

  // --- Get Windows version using RtlGetVersion (true version) ---
```

```
    HMODULE hMod = GetModuleHandleA("ntdll.dll");
    if (!hMod)
        return 3;

    RtlGetVersionPtr fn = (RtlGetVersionPtr)GetProcAddress(hMod, "RtlGetVersion");
    if (!fn)
        return 4;

    RTL_OSVERSIONINFOW ver = { 0 };
    ver.dwOSVersionInfoSize = sizeof(ver);

    if (fn(&ver) != 0)
        return 5;

    // Format OS version string
    // Example: "Windows 10 (10.0.19045)"
    int written = snprintf(
        osBuf, osBufSize,
        "Windows %lu.%lu (Build %lu)",
        ver.dwMajorVersion,
        ver.dwMinorVersion,
        ver.dwBuildNumber
    );

    if (written < 0 || (size_t)written >= osBufSize)
        return 6; // insufficient buffer

    return 0; // success
}

int platformInfo()
{
    return 0;
}
```