CVE 2025 25567: A Stack-Based Buffer Overflow on SoftEtherVPN

Jordan Day, Shanzay Khan, Alexandra Kotsinyan

Introduction

Summary

We found this exploit from the CVE website. It was a buffer overflow exploit written by Drake C that was exploitable from a specific file in SoftEther VPN. We utilize the vpncmd binary file and give it a crafted input of 0x89 (137 decimal) bytes to overwrite the instruction pointer (RIP). The vulnerability exists in the Internat.c file.

The vulnerability has been patched in the stable release and as of writing this, only exists in the unstable version.

Target: SoftEtherVPN

SoftEther VPN is an open source and multi-protocol VPN designed to allow users to create VPN connections and give secure communication channels for data transfer via strong encryption. The code is available to the public, and as such can be analyzed to look for potential vulnerabilities.

Bug: Stack-Based Buffer Overflow

There is a vulnerability within a file, src/Mayaqua/Internat.c, at lines 2458 to 2503. Two functions, UniToInt and UniToStrForSingleChars, do not have bounds checking for the tmp variable. This allows us to create a buffer overflow. By attacking this, an attacker can overtake the instruction pointer(RIP), and potentially gain control of program execution. The bug is run via the build/vpncmd binary file, which interacts with code from the src/Mayaqua folder. This folder contains Internat.c–which has the two exploitable functions for the bug as explained above– and Str.c which has the two functions utilized in Internat.c; ToInti and ToInt.

Exploit Development and Execution

We initially began working on creating a stack diagram and confirming it with the original CVE writeup using x86_64 machines. We found that the buffer length was 0x80 (128 in decimal) and that the register RBX was also on the stack frame for our entry point, UniToInt. We found that consistent with the CVE writeup, sending any more than 0x88 (136 decimal) bytes would start to overflow RIP, our instruction pointer.

Program Flow

The general program flow, relevant to our exploit, is as follows:

1. The vpncmd binary receives user input.
2. The input is then processed by functions in Internat.c
    a. It first is passed through UniToInt.
    b. Then, it is passed through UniToStrForSingleChars, returning to UniToInt.
    c. UniToInt then passes the input to functions in Str.c.
3. Within Str.c, the input is passed through several functions.
    a. The first function it is passed through is ToInti.
    b. Then, the input is passed to ToInt, which calls the C language stdlib function, strtoul.

A description of each function is as follows:

- UniToInt: Converts a string to an integer, with error checks to ensure no null input is given

```C/C++
// Convert a string to an integer
UINT UniToInt(wchar_t *str)
{
	char tmp[128];
	// Validate arguments
	if (str == NULL)
	{
		return 0;
	}

	UniToStrForSingleChars(tmp,
	sizeof(tmp), str);

	return ToInti(tmp);
}
```

- UniToStrForSingleChars: converts only single-byte characters in the Unicode string to a char string. Has a bounds check to ensure values stay between 0 - 0xff

```C/C++
// Convert only single-byte characters
in the Unicode string to a char string
void UniToStrForSingleChars(char *dst,
UINT dst_size, wchar_t *src)
{
	UINT i;
	// Validate arguments
	if (dst == NULL || src == NULL)
	{
	return;
	}

	for (i = 0;i < UniStrLen(src) +
1;i++)
	{
	wchar_t s = src[i];
	char d;

	if (s == 0)
	{
		d = 0;
	}
	else if (s <= 0xff)
	{
		d = (char)s;
	}
	else
	{
		d = ' ';
	}

	dst[i] = d;
	}
}
```

- ToInti: converts a string to a signed integer

```C/C++
// Convert the string to a signed
integer
int ToInti(char *str)
{
	// Validate arguments
	if (str == NULL)
	{
		return 0;
	}

	return (int)ToInt(str);
}
```

- ToInt: converts a string to an integer

```C/C++
// Convert a string to an integer
UINT ToInt(char *str)
{
	// Validate arguments
	if (str == NULL) {
		return 0;
	}

	// Ignore the octal literal
	while (true) {
		if (*str != '0') {
			break;
		}
		if ((*(str + 1) == 'x')
	|| (*(str + 1) == 'X')) {
			break;
		}
		str++;
	}
```

```
return (UINT)strtoul(str, NULL,
0);
```

Control of RIP

To get control of RIP, in order for strtoul to not error out and cause our program to exit, we must give a valid base-10 integer first. Then, we fill up the rest of our 0x80 buffer size (tmp) with ASCII characters (the reason why we specifically do ASCII characters is explained below in the “Limitations” section). The next 8 bytes we send will overflow the RBX register, which is next on the stack. Here, we replace RBX with identifiable dummy characters (that are also ASCII, explained in “Limitations”), that way we can easily tell when we’ve overflowed into RBX.

Once we’ve overflowed all of the bytes in RBX, the next bytes we provide the program will overwrite into RIP, the instruction pointer that tells the computer which instruction to execute next.

Limitations

We see two primary paths to a meaningful exploitation once we have control of RIP: first, to utilize a ROP chain; second, to execute somewhere existing in memory.

First, regarding a ROP chain approach, we found this to be impossible within our time frame due to the fact that this program utilizes address canonicalization[1]. This means that we are unable to start a ROP chain because we would need to utilize the first four bytes of our memory address to get a gadget that would function as a stack pivot. Unfortunately, we have discovered that the program will only accept input that contains ASCII characters that are *not* Unicode characters. This means that we must provide bytes within the range 0x20 to 0x7E, inclusive. Due to address canonicalization, the first and second bytes must be either 0x00 or 0xFF, both of which are outside of those bounds.

Then, that leaves us with the approach of finding and executing a memory address. Because all memory addresses in this program must be canonicalized, that is not a problem in this approach. However, we do still need to find an address where every byte is within our ASCII bounds. After running the command “info proc mapping” in gdb after the dynamic libraries were all loaded in, we found that there were no memory locations that were both executable and within our bounds, so we were unable to find a valid memory address.

[1] See Intel® 64 and IA-32 Architectures Software Developer’s Manual section 3.3.7.1 on Canonical Addressing

Mitigation, a Possible Solution

C/C++

```
// Convert only single-byte characters
in the Unicode string to a char string
void UniToStrForSingleChars(char *dst,
UINT dst_size, wchar_t *src)
{
    UINT i;

    // Validate arguments
    if (dst == NULL || src == NULL) {
        return;
    }
    //MITIGATION: if the size of input
is greater than size of tmp,
    //... then cap input at sizeof(tmp)
    UINT writeLength;
```

```
    UINT srcLength = UniStrLen(src) +
1;
    if (srcLength > dst_size) {
        writeLength = dst_size;
    } else {
        writeLength = srcLength;
    }

    for (i = 0; i < writeLength ; i++)
    {
        wchar_t s = src[i];
```

Our proposed solution to the buffer overflow exploit. The idea behind it is if the size of the input given in the function UniToStrForSingleChar is greater than the size of tmp (what our exploit has been built off of), then adding a checker to cap the actual size of the input given at sizeof(tmp) would stop the buffer overflow from happening, since there would now be a cap that cannot be overflowed past as easily.

Contributions

Researchers

Jordan Day selected the CVE target. **Alexandra Kotsinyan** created the virtual machines and exploit environment for the group. **Shanzay Khan** did exploit research on SoftEtherVPN and stack overflow bugs. **Jordan Day** did the initial reversing of the program and the exploit development. **Shanzay Khan** and **Jordan Day** wrote the paper report (equal contribution). **Alexandra Kotsinyan** wrote the mitigation.

Acknowledgements

This report is made with thanks to Drake C., who found the vulnerability and provided a short writeup that gave us a good starting point. We would also like to thank Haylin Moore, who assisted us in the reversing and debugging process and provided us great moral support.