Why is my shellcode being corrupted?

allelesecurity.com/why-is-my-shellcode-being-corrupted

By Allele Security Intelligence June 10, 2026



In 2022, while one of our researchers was developing our Linux Binary Exploitation training, he approached me with a curious issue. During the exploitation of the simplest case of a stack-based buffer overflow, the shellcode was being corrupted on GDB. It took us a while to figure out what was happening, but we understood it. At least, we thought we had understood.

I revisited that topic this week to teach the students of our training about that case and I noticed something that I hadn't noticed back then. I noticed that the behavior he faced was an illusion! I then started to write this blog post aiming to show the illusion GDB creates and in the middle of it, when experimenting on GDB, I came across weird behaviors that I didn't understand immediately.

That journey led me to discover much more about GDB than I knew when I started this blog post. The main points I discovered about GDB are that a breakpoint hit is not always caused by hardware or software breakpoints, and that GDB has a feature called Displaced Stepping (or Out-Of-Line execution) that hides the execution of instructions from the user.

The issue

To demonstrate the issue, we followed a specific chronological flow:

1. Set a software breakpoint at the target address *before* execution begins.
2. Run the binary, passing a shellcode as a command-line argument.
3. Allow the binary's vulnerable function to copy the shellcode onto the stack, overwriting the saved return address.

(Note: For the sake of simplicity in this blog post, our shellcode consists entirely of NOP instructions).

If you follow this exact workflow, you will notice that the shellcode becomes corrupted after the execution phase. In GDB, when examining a memory address as a multi-byte word (like `x/gx`), Little-Endian representation causes the lowest address byte to appear on the far right of the value. In that case, the first byte on the address (0x7fffffffe050) that we set a breakpoint, shows 0x00 rather than 0x90, as shown below:

[User-mode GDB]



```
(gdb) b *0x7fffffffe050
Breakpoint 1 at 0x7fffffffe050
(gdb) r `perl -e 'print "\x90"x264 . "\x50\xe0\xff\xff\xff\x7f"'`
Starting program: /home/user/work/linux-binary-exploitation-exercises-development/exer
...
buffer: 0x7fffffffe050
Hello ������������������������������������������������������������������������������������

Program received signal SIGSEGV, Segmentation fault.
0x00007fffffffe159 in ?? ()
=> 0x00007fffffffe159:  00 00   add    %al,(%rax)
(gdb) x/10i 0x7fffffffe050
   0x7fffffffe050:      add    %dl,-0x6f6f6f70(%rax)
...
   0x7fffffffe05e:      nop
(gdb) x/2gx 0x7fffffffe050
0x7fffffffe050: 0x9090909090909000      0x9090909090909090
(gdb)
```

We had inserted only NOPs (0x90), but when dumping the shellcode from the stack, after the code crashed, we see one of the NOPs has been replaced by a NULL byte (0x00). What has happened?

The explanation

Even though I hadn't faced the issue before, it was simple to figure out given a basic understanding of how debuggers work. The problem demonstrated is well known for experienced professionals [1]. Debuggers work basically in two ways: using hardware debug registers or by replacing the targeted code instruction with a breakpoint instruction (int3, opcode 0xcc). The first method, although usable by GDB, is not the most common way of defining breakpoints in binaries (user-mode debugging), so we stick to the second.

By replacing the targeted code with a breakpoint instruction, when it is executed, an exception is triggered and the CPU stops the execution, the kernel handles it and returns control to the debugger. This is how debuggers work and more details can be found in [2]. Although the code is not self-modifying, setting a breakpoint on writable memory and having issues is a known pattern affecting debuggers.

When the breakpoint is set, the debugger saves the current byte at the breakpoint address (0x00 because the stack is uninitialized) and inserts a breakpoint instruction there. We can see this happening if we enable target debugging (`set debug target 1`) on GDB. In the output below, the saved byte is shown as `00`. In other environments, the stack might have a different byte at the breakpoint address.

[User-mode GDB]

```
Breakpoint 1, 0x000000000040117e in main ()
=> 0x000000000040117e <main+8>: 48 81 ec 10 01 00 00    sub    $0x110,%rsp
(gdb) x/2gx 0x7fffffffe050
0x7fffffffe050: 0x0000000000ff0000      0x0000000000000000
(gdb) set debug target 1
(gdb) b *0x7fffffffe050
...
<- multi-thread->xfer_partial (1, (null), 0x7fffffffd9a0, 0x0, 0x7fffffffe050, 0x1, 0x
multi-thread:target_xfer_partial (1, (null), 0x7fffffffd9a0, 0x0, 0x7fffffffe050, 1) =
 00
Breakpoint 2 at 0x7fffffffe050
(gdb)
```

Below, just to demonstrate a different original byte being saved internally by GDB, we set a breakpoint on the ret (0xc3) instruction at the address 0x0401228. In the output below, the saved byte is shown as `c3`.

[User-mode GDB]

```
(gdb) b main
Breakpoint 1 at 0x40117e
(gdb) r
...
Breakpoint 1, 0x000000000040117e in main ()
=> 0x000000000040117e <main+8>: 48 81 ec 10 01 00 00    sub    $0x110,%rsp
(gdb) x/i 0x0401228
   0x401228 <main+178>: ret
(gdb) x/2gx 0x401228
0x401228 <main+178>:    0xfa1e0ff3000000c3      0x08c4834808ec8348
(gdb) set debug target 1
(gdb) b *0x401228
...
<- multi-thread->xfer_partial (1, (null), 0x7fffffffd9a0, 0x0, 0x401228, 0x1, 0x1) = 1
multi-thread:target_xfer_partial (1, (null), 0x7fffffffd9a0, 0x0, 0x401228, 1) = 1, 1,
 c3
Breakpoint 2 at 0x401228
(gdb)
```

When analyzing this behavior, the configuration of GDB plays a critical role. With the default setting (`set breakpoint always-inserted off`), GDB handles memory states in stages:

1. **Breakpoint Definition:** GDB reads the target memory address, saves the original byte (`0x00`) to an internal cache, and substitutes it with an `int3` (`0xcc`) instruction.
2. **Shellcode Injection:** The code copies the shellcode to the stack and overwrites the `0xcc` trap instruction with NOPs. The CPU continues to execute instructions down the stack perfectly.
3. **The Stop/Crash Sequence:** The application eventually segments faults further along the stack. When execution stops, GDB recovers the original byte, pasting its cached `0x00` byte back over our written NOP instruction.

But why is the shellcode not crashing on the corrupted instruction or even hitting a breakpoint?

The illusion

Thinking about that question, I noticed that it is an illusion. If the shellcode is really corrupted, the instruction executed is the one shown in the disassembly below:

[User-mode GDB]

```
0x7fffffffe050: add    %dl,-0x6f6f6f70(%rax)
```

That instruction should crash because `$RAX` doesn't contain any valid address for that instruction to execute successfully. In the same way, the breakpoint is also not being hit. If we pay attention, the code is crashing on:

[User-mode GDB]



```
=> 0x00007fffffffe159:  00 00   add    %al,(%rax)
```

That is after the NOPs (shellcode) are executed. So, what is happening?

The answer is that it is complex and there are a lot of factors involved. It depends on how GDB is configured, as we will see, and the runtime configuration, like where in the code the breakpoints are being set. We will cover most of them here.

When we insert the shellcode on the stack after the breakpoint was added there, we overwrite the breakpoint instruction added by the debugger with our NOPs. This is why the breakpoint is not being hit. This makes sense, but I wanted to make sure the theory was right, that it's an illusion shown by GDB, and decided to get evidence of it by looking at the physical page of the instruction. This way, we could see the memory beyond what GDB shows, when the code skips the breakpoint, executes all the NOPs, and crashes in an unrelated instruction as shown above.

We can easily get the physical page of an address and read it through the `crash` utility or with GDB through "`monitor phys`" when it is attached to VMware (kernel debugging). You can also do this with GDB attached to QEMU. But when we did it, the shellcode really seemed corrupted.

[User-mode Crash]



```
crash> vtop -c 2787 0x7fffffffe050
VIRTUAL     PHYSICAL
7fffffffe050  17ebb2050

   PGD: 10c15e7f8 => 10e599067
   PUD: 10e599ff8 => 116544067
   PMD: 116544ff8 => 105f7c067
   PTE: 105f7cff0 => 17ebb2867
  PAGE: 17ebb2000
```

```
   PTE     PHYSICAL   FLAGS
17ebb2867  17ebb2000  (PRESENT|RW|USER|ACCESSED|DIRTY)

      VMA           START       END     FLAGS FILE
ffff88810384f0d0 7ffffffde000 7fffffff000 100177

      PAGE        PHYSICAL      MAPPING       INDEX CNT FLAGS
ffffea0005faec80 17ebb2000 ffff88810665fb61 7fffffffe  1 17ffffc008003e referenced,upt
crash> rd -p 17ebb2050
       17ebb2050:  9090909090909000                    ........
crash>
```

[Kernel Debugger — VMware]

```
(gdb) monitor phys
(gdb) x/2gx 0x17ebb2050
0x17ebb2050:    0x9090909090909000      0x9090909090909090
(gdb)
```

We can see the physical page starts with `0x00` and not `0x90`. This didn't make sense initially. If the shellcode was really corrupted, why didn't it crash then? I did some experiments to understand it a bit better.

Recapitulating: if we don't set any breakpoint on the code in the `main()` function, especially after writing the shellcode on the stack (via the `strcpy()` function, as this is a classic buffer overflow exercise), the shellcode executes fine and is not corrupted. The code crashes after the shellcode is executed successfully. But, when we inspect the shellcode through the physical page, it really is corrupted.

That behavior is an illusion because when inspecting the shellcode after the program crashes, GDB re-inserts the original byte where the breakpoints were defined. That's how it works independently of any configuration.

Otherwise, if the shellcode were really corrupted during execution, it should have crashed. We confirm this by setting a breakpoint on the address in the kernel debugger and executing the binary again. This time, we defined two breakpoints on the shellcode, and in both GDB sessions.

[User-mode GDB]

```
(gdb) set breakpoint auto-hw off
(gdb) set breakpoint always-inserted off
(gdb) b *0x00007fffffffe050
Breakpoint 1 at 0x7fffffffe050
(gdb) b *0x00007fffffffe060
Breakpoint 2 at 0x7fffffffe060
(gdb) r `perl -e 'print "\x90"x264 . "\x50\xe0\xff\xff\xff\x7f"'`
Starting program: /home/user/work/linux-binary-exploitation-exercises-development/exer
...
buffer: 0x7fffffffe050
Hello ������������������������������������������������������������������������������������
Program received signal SIGSEGV, Segmentation fault.
0x00007fffffffe159 in ?? ()
=> 0x00007fffffffe159:  00 00   add    %al,(%rax)
(gdb) x/15i 0x7fffffffe050
   0x7fffffffe050:      add    %dl,-0x6f6f6f70(%rax)
   0x7fffffffe056:      nop
...
   0x7fffffffe05f:      nop
   0x7fffffffe060:      add    %dl,-0x6f6f6f70(%rax)
...
   0x7fffffffe068:      nop
(gdb)
```

[Kernel Debugger — VMware]

```
(gdb) info breakpoints
Num     Type           Disp Enb Address            What
1       hw breakpoint  keep y   0x00007fffffffe050
        breakpoint already hit 23 times
2       hw breakpoint  keep y   0x00007fffffffe060
        breakpoint already hit 13 times
(gdb) c
Continuing.
[Switching to Thread 3]

Thread 3 hit Breakpoint 1, 0x00007fffffffe050 in ?? ()
=> 0x00007fffffffe050:  90                      nop
(gdb) c
Continuing.

Thread 3 hit Breakpoint 2, 0x00007fffffffe060 in ?? ()
=> 0x00007fffffffe060:  90                      nop
(gdb) c
Continuing.
```

As we can see, the shellcode is fine. It looks corrupted after the GDB stops when the code crashes, but when it was executed, as we confirmed by setting two breakpoints on it via GDB attached to the VMware, the addresses contained a NOP instruction as expected. This confirms the illusion performed by GDB.

Re-inserting the breakpoints

The runtime configuration of GDB is really important to understand how it works. I experimented initially with the configurations below and enabled "`breakpoint always-inserted`" later.

[User-mode GDB]

```
(gdb) set breakpoint auto-hw off
(gdb) set breakpoint always-inserted off
```

The configuration "breakpoint always-inserted" off is the default on our systems. In that case, GDB removes and re-inserts the breakpoints every time it stops. That's why if we stop the code right after it is copied to the stack, and inspect it and its physical memory, we really see it as corrupted, and this time, it crashes. It's not an illusion.

When the code stops before finishing, GDB re-inserts the breakpoint and this time, it overwrites the NOP with the breakpoint instruction. That's why the breakpoints on the shellcode are hit. After hitting it, GDB replaces the NOP with the original byte saved, the NULL byte and this really corrupts the shellcode.

In order to see that happening, we add a third breakpoint at the instruction after the shellcode has been copied to the stack. Now, at that time, the breakpoint instructions added on the addresses 0x7fffffffe050 and 0x7fffffffe060 are not overwritten. Despite the breakpoints being re-inserted when the code resumes execution, the original value saved from those breakpoints addresses are still a NULL byte.

[User-mode GDB]

```
(gdb) r
...
Starting program: /home/user/work/linux-binary-exploitation-exercises-development/exer
...
```

```
buffer: 0x7fffffffe050

Breakpoint 3, 0x0000000000401204 in main ()
=> 0x0000000000401204 <main+142>:       48 8d 85 00 ff ff ff    lea    -0x100(%rbp),%ra
(gdb) x/15i 0x7fffffffe050
   0x7fffffffe050:      add    %dl,-0x6f6f6f70(%rax)
   0x7fffffffe056:      nop
...
   0x7fffffffe05f:      nop
   0x7fffffffe060:      add    %dl,-0x6f6f6f70(%rax)
...
   0x7fffffffe068:      nop
(gdb) c
Continuing.
Hello ████████████████████████████████████████████████████████████████████████████████████████████████

Breakpoint 1, 0x00007fffffffe050 in ?? ()
=> 0x00007fffffffe050:  00 90 90 90 90 90       add    %dl,-0x6f6f6f70(%rax)
(gdb) c
Continuing.

Program received signal SIGSEGV, Segmentation fault.
0x00007fffffffe050 in ?? ()
=> 0x00007fffffffe050:  00 90 90 90 90 90       add    %dl,-0x6f6f6f70(%rax)
(gdb)
```

We confirm the NOP instruction was replaced by the breakpoint instruction through kernel debugging.

[Kernel Debugger — VMware]

```
(gdb) c
Continuing.
[Switching to Thread 2]

Thread 2 hit Breakpoint 1, 0x00007fffffffe050 in ?? ()
=> 0x00007fffffffe050:  cc                     int3
(gdb) c
Continuing.
```

It is not easy to demonstrate that the breakpoints are removed when the program stops and re-inserted when it resumes execution, but if we enable "`set debug target on`", we can at least see the breakpoint functions `remove_breakpoint()` and `insert_breakpoint()` being called on the addresses `0x00007fffffffe050` and `0x00007fffffffe060`. Since GDB 14 [3][4] you can also enable "`set debug breakpoint on`". This configuartion allows you to see the

breakpoints being added and removed, including the breakpoints GDB automatically adds behind the user.

[User-mode GDB]

```
(gdb) r
...
Starting program: /home/user/work/linux-binary-exploitation-exercises-development/exer
...
<- multi-thread->xfer_partial (1, (null), 0x0, 0x5555565f9eac, 0x7fffffffe050, 0x1, 0x
multi-thread:target_xfer_partial (2, (null), 0x0, 0x5555565f9eac, 0x7fffffffe050, 1) =
<- multi-thread->remove_breakpoint (i386:x86-64, 0x00007fffffffe050, 0) = 0
...
<- multi-thread->xfer_partial (1, (null), 0x0, 0x5555565fa05c, 0x7fffffffe060, 0x1, 0x
multi-thread:target_xfer_partial (2, (null), 0x0, 0x5555565fa05c, 0x7fffffffe060, 1) =
<- multi-thread->remove_breakpoint (i386:x86-64, 0x00007fffffffe060, 0) = 0
...
Breakpoint 3, -> multi-thread->get_unwinder (...)
...
0x0000000000401204 in main ()
=> 0x0000000000401204 <main+142>:       -> multi-thread->xfer_partial (...)
...
48 8d 85 00 ff ff ff    lea    -0x100(%rbp),%rax
...
(gdb)
```

[User-mode GDB]

```
(gdb) c
-> multi-thread->log_command (...)
<- multi-thread->log_command (c)
Continuing.
...
<- multi-thread->xfer_partial (1, (null), 0x0, 0x555555c71b60, 0x7fffffffe050, 0x1, 0x
multi-thread:target_xfer_partial (2, (null), 0x0, 0x555555c71b60, 0x7fffffffe050, 1) =
 cc
<- multi-thread->insert_breakpoint (i386:x86-64, 0x00007fffffffe050) = 0
...
<- multi-thread->xfer_partial (1, (null), 0x7fffffffd610, 0x0, 0x7fffffffe060, 0x1, 0x
multi-thread:target_xfer_partial (1, (null), 0x7fffffffd610, 0x0, 0x7fffffffe060, 1) =
 00
-> multi-thread->xfer_partial (...)
<- multi-thread->xfer_partial (1, (null), 0x0, 0x555555c71b60, 0x7fffffffe060, 0x1, 0x
```

```
multi-thread:target_xfer_partial (2, (null), 0x0, 0x555555c71b60, 0x7fffffffe060, 1) =
 cc
<- multi-thread->insert_breakpoint (i386:x86-64, 0x00007fffffffe060) = 0
...
```

We can't easily see the original byte saved (shadow content) for specific breakpoints when they are inserted, but they are shown on the output above: "0xcc" and "0x00". This confirms that once they are saved, they will be used for those specific breakpoints independent of the addresses content when the breakpoint are re-inserted. That's the reason the shellcode is corrupted when the program stops and resumes.

Enabling breakpoint always-insert

Now, after enabling "`breakpoint always-inserted on`", the breakpoints aren't removed and re-inserted every time the code stops, but GDB still keeps showing the original byte saved when the memory is inspected. If we define that configuration and enable the same debug option above, we no longer see the remove_breakpoint() and insert_breakpoint() functions being called when the code stops. As the breakpoints aren't re-inserted every time the code stops, the shellcode now runs without triggering them.

With our third breakpoint set right after the shellcode is copied to the stack, we can inspect the physical page and it shows the shellcode is intact. Neither is the breakpoint hit, nor does the shellcode crash. Once the breakpoint is defined, the breakpoint instruction should be inserted and once overwritten, GDB doesn't re-insert it.

[User-mode GDB]



```
(gdb) set breakpoint always-inserted on
(gdb) r
...
Starting program: /home/user/work/linux-binary-exploitation-exercises-development/exer
...
buffer: 0x7fffffffe050

Breakpoint 4, 0x0000000000401204 in main ()
=> 0x0000000000401204 <main+142>:       48 8d 85 00 ff ff ff    lea    -0x100(%rbp),%ra
(gdb)
```

[User-mode Crash]



```
crash> vtop -c 3272 0x7fffffffe050
VIRTUAL     PHYSICAL
7fffffffe050  181ecb050

   PGD: 1038f67f8 => 10dd8b067
   PUD: 10dd8bff8 => 10c169067
   PMD: 10c169ff8 => 10aafc067
   PTE: 10aafcff0 => 181ecb867
  PAGE: 181ecb000

   PTE     PHYSICAL   FLAGS
181ecb867  181ecb000  (PRESENT|RW|USER|ACCESSED|DIRTY)

      VMA           START       END     FLAGS FILE
ffff88810c16dd00 7ffffffde000 7ffffffff000 100177

      PAGE        PHYSICAL      MAPPING       INDEX CNT FLAGS
ffffea000607b2c0 181ecb000 ffff888103ab5549 7fffffffe  1 17ffffc008003e referenced,upt
crash> rd -p 181ecb050
       181ecb050:  9090909090909090                    ........
crash>
```

GDB maintains the illusion the shellcode is corrupted, but the code executes fine.

[User-mode GDB]

```
(gdb) r
...
Starting program: /home/user/work/linux-binary-exploitation-exercises-development/exer
...
buffer: 0x7fffffffe050

Breakpoint 4, 0x0000000000401204 in main ()
=> 0x0000000000401204 <main+142>:       48 8d 85 00 ff ff ff    lea    -0x100(%rbp),%r
(gdb) x/20i 0x7fffffffe050
   0x7fffffffe050:      add    %dl,-0x6f6f6f70(%rax)
...
   0x7fffffffe05f:      nop
   0x7fffffffe060:      add    %dl,-0x6f6f6f70(%rax)
...
   0x7fffffffe06d:      nop
(gdb)
```

Below we see the crash didn’t happen because of the shellcode.

[User-mode GDB]



```
(gdb) set breakpoint always-inserted on
(gdb) r
...
Starting program: /home/user/work/linux-binary-exploitation-exercises-development/exer
...
buffer: 0x7fffffffe050

Breakpoint 4, 0x0000000000401204 in main ()
=> 0x0000000000401204 <main+142>:       48 8d 85 00 ff ff ff    lea    -0x100(%rbp),%r
(gdb) c
Continuing.
Hello ████████████████████████████████████████████████████████████████████████████████
Program received signal SIGSEGV, Segmentation fault.
0x00007fffffffe159 in ?? ()
=> 0x00007fffffffe159:  00 00   add    %al,(%rax)
(gdb) x/15i 0x7fffffffe050
   0x7fffffffe050:      add    %dl,-0x6f6f6f70(%rax)
   0x7fffffffe056:      nop
   0x7fffffffe057:      nop
   0x7fffffffe058:      nop
   0x7fffffffe059:      nop
   0x7fffffffe05a:      nop
   0x7fffffffe05b:      nop
   0x7fffffffe05c:      nop
   0x7fffffffe05d:      nop
   0x7fffffffe05e:      nop
   0x7fffffffe05f:      nop
   0x7fffffffe060:      add    %dl,-0x6f6f6f70(%rax)
   0x7fffffffe066:      nop
   0x7fffffffe067:      nop
   0x7fffffffe068:      nop
(gdb)
```

After enabling the configuration “breakpoint always-inserted on”, we restore the same behavior we see initially, but this time we are allowed to check the physical memory.

This shows that, by default, even when the breakpoints are removed and re-inserted, the shadow content saved internally continues the same. This is the reason breakpoints on self-modifying code or on writable memory is tricky.

But the journey doesn’t stop here. When we moved the breakpoint from the instruction right after strcpy() to the ret instruction before jumping to the shellcode, the behavior changed

completely.

The debugger quirk

Writing this post and performing some experiments when the configuration “breakpoint always-inserted” is enabled, we discovered interesting behaviors. Depending on where the third breakpoint in the code is set, we see a different behavior in GDB. In the case above, the third breakpoint is defined right after the `strcpy()` function call that copies the shellcode to the stack so we could inspect it. In practice, it didn’t happen that way. I first set the third breakpoint on the `ret` instruction right before jumping to the shellcode, but I saw weird things happening and I couldn’t explain immediately.

[User-mode GDB]

```
(gdb) set breakpoint always-inserted on
(gdb) info breakpoints
Num     Type           Disp Enb Address            What
1       breakpoint     keep y   0x00007fffffffe050
2       breakpoint     keep y   0x00007fffffffe060
4       breakpoint     keep y   0x0000000000401228 <main+178>
(gdb) x/i 0x0000000000401228
   0x401228 <main+178>: ret
(gdb) r
...
Starting program: /home/user/work/linux-binary-exploitation-exercises-development/exer
...
buffer: 0x7fffffffe050
Hello ����������������������������������������������������������������������������������

Breakpoint 4, 0x0000000000401228 in main ()
=> 0x0000000000401228 <main+178>:       c3      ret
(gdb) c
Continuing.

Breakpoint 1, 0x00007fffffffe050 in ?? ()
=> 0x00007fffffffe050:  00 90 90 90 90 90       add    %dl,-0x6f6f6f70(%rax)
(gdb) c
Continuing.

Program received signal SIGSEGV, Segmentation fault.
0x00007fffffffe050 in ?? ()
=> 0x00007fffffffe050:  00 90 90 90 90 90       add    %dl,-0x6f6f6f70(%rax)
(gdb)
```

In this case, the breakpoint on the shellcode is being hit and the shellcode really seems to be corrupted. But in the other case, when setting the breakpoint right after strcpy() and enabling “breakpoint always-inserted on”, we don’t hit the breakpoint on the shellcode nor it crashes. We only changed the breakpoint on the function main() a bit further.

Keeping the configuration and changing the breakpoint location, besides hitting the breakpoint, it crashes. It gets even weirder because we can’t see any instruction being executed in the kernel debugger. The breakpoint below is not hit when the code above is executed.

[Kernel Debugger — VMware]

```
(gdb) info breakpoints
Num     Type           Disp Enb Address            What
1       hw breakpoint  keep y   0x00007fffffffe050
(gdb) c
Continuing.
```

Let’s check the physical page of the shellcode when the breakpoint on the ret instruction is hit.

[User-mode GDB]

```
(gdb) r
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /home/user/work/linux-binary-exploitation-exercises-development/exer
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".
buffer: 0x7fffffffe050
Hello ������������������������������������������������������������������������������������

Breakpoint 3, 0x0000000000401228 in main ()
=> 0x0000000000401228 <main+178>:       c3      ret
(gdb)
```

[User-mode Crash]

```
crash> vtop -c 3282 0x7fffffffe050
VIRTUAL     PHYSICAL
```

```
7fffffffe050  133a70050

   PGD: 1038f67f8 => 10374c067
   PUD: 10374cff8 => 103859067
   PMD: 103859ff8 => 10dd83067
   PTE: 10dd83ff0 => 133a70867
  PAGE: 133a70000

   PTE     PHYSICAL   FLAGS
133a70867  133a70000  (PRESENT|RW|USER|ACCESSED|DIRTY)

      VMA           START       END     FLAGS FILE
ffff88810aa85680 7ffffffde000 7fffffffe000 100177

      PAGE        PHYSICAL      MAPPING       INDEX CNT FLAGS
ffffea0004ce9c00 133a70000 ffff888103ab5549 7fffffffe  1 17ffffc008003e referenced,upt
crash> rd -p 133a70050
       133a70050:  9090909090909090                    ........
crash>
```

The shellcode seems intact, but it still crashes when executed.

[User-mode GDB]

```
(gdb) r
...
Starting program: /home/user/work/linux-binary-exploitation-exercises-development/exer
...
buffer: 0x7fffffffe050
Hello ████████████████████████████████████████████████████████████████████████████████

Breakpoint 3, 0x0000000000401228 in main ()
=> 0x0000000000401228 <main+178>:       c3      ret
(gdb) c
Continuing.

Breakpoint 1, 0x00007fffffffe050 in ?? ()
=> 0x00007fffffffe050:  00 90 90 90 90 90       add    %dl,-0x6f6f6f70(%rax)
(gdb) c
Continuing.

Program received signal SIGSEGV, Segmentation fault.
0x00007fffffffe050 in ?? ()
=> 0x00007fffffffe050:  00 90 90 90 90 90       add    %dl,-0x6f6f6f70(%rax)
(gdb)
```

It seems the instruction is being executed because we receive a `SIGSEGV`, but there is no trace of it really being executed by the CPU.

In summary, depending on where the third breakpoint is placed, GDB can produce a breakpoint hit and apparent shellcode corruption even when the physical memory is intact and no instruction was ever executed at that address — a behavior that turns out to be rooted in how GDB handles single-stepping internally.

After a deep dive into GDB and its internals, we learned some things. We learned that the `continue` command in GDB is, in fact, a step-over and a call to `ptrace()` with the `PTRACE_CONT` argument. This is needed because of multi-threaded environments, and the real implementation depends on another GDB configuration: `all-stop` mode.

That configuration dictates if all other threads are stopped or if they are allowed to keep running. Because the breakpoint instruction needs to be bypassed after it is hit and the `continue` command is issued, if the debugger removes the breakpoint instruction and continues the execution of the process, other non-stopped threads executing the same code will miss the breakpoint. To solve this, GDB implements something called Displaced Stepping, but before getting into it, let's explain why in that case the breakpoint is hit.

When GDB executes the step-over internally due to the continue command, it simply checks if the next instruction is on the list of breakpoint addresses. If this is the case, the breakpoint is hit even when there's no breakpoint instruction there as we overwrite it with our NOPs since the beginning of this post. This is the first behavior of GDB that we didn't expect. We confirmed this by checking the GDB's source code.

File: binutils-gdb/gdb/infrun.c

```
6960 static void
6961 handle_signal_stop (struct execution_control_state *ecs)
6962 {
...
7072   /* Pull the single step breakpoints out of the target.  */
7073   if (ecs->event_thread->stop_signal () == GDB_SIGNAL_TRAP)
7074     {
7075       struct regcache *regcache;
7076       CORE_ADDR pc;
...
7078       regcache = get_thread_regcache (ecs->event_thread);
7079       const address_space *aspace = ecs->event_thread->inf->aspace.get ();
...
```

```
7081       pc = regcache_read_pc (regcache);
...
7083       /* However, before doing so, if this single-step breakpoint was
7084          actually for another thread, set this thread up for moving
7085          past it.  */
7086       if (!thread_has_single_step_breakpoint_here (ecs->event_thread,
7087                                                    aspace, pc))
7088         {
7089           if (single_step_breakpoint_inserted_here_p (aspace, pc))
7090             {
...
7094               ecs->hit_singlestep_breakpoint = 1;
7095             }
7096         }
7097       else
7098         {
...
7101         }
7102     }
...
7465 }
```

File: binutils-gdb/gdb/breakpoint.c



```
14059 /* Check whether a software single-step breakpoint is inserted at
14060    PC.  */
...
14062 int
14063 single_step_breakpoint_inserted_here_p (const address_space *aspace,
14064                                         CORE_ADDR pc)
14065 {
14066   for (breakpoint &bpt : all_breakpoints ())
14067     {
14068       if (bpt.type == bp_single_step
14069           && breakpoint_has_location_inserted_here (&bpt, aspace, pc))
14070         return 1;
14071     }
14072   return 0;
14073 }
```

The function `handle_signal_stop()` is likely called because the step-over is implemented using hardware single-stepping, so a `SIGTRAP` signal is delivered to the process. This explains why in this case the breakpoint is triggered: it is basically because there is a breakpoint for the next instruction according to GDB's internal breakpoint list, and GDB will stop on it even though there is no actual breakpoint instruction there. It's like a virtual breakpoint!

We validated this behavior by setting two breakpoints on two adjacent instructions, removing the breakpoint instruction from the second instruction, and hitting the first one. When single-stepping, the second breakpoint was triggered even with no breakpoint instruction present in memory. It is a “coincidence”, during the single-step, GDB internally notices the next instruction pointer matches an entry on its breakpoint list and triggers a break. I expected hardware and software breakpoints were the only way a breakpoint could be hit, but learned this is not the case. It simply breaks if RIP is on the internal breakpoint list.

We now know why the breakpoint on the shellcode is being hit during the `ret` instruction execution, but we still don’t know why we can’t see the corrupted shellcode being executed, or why we don’t see it as corrupted in physical memory. The answer to this is the Displaced Stepping feature.

The GDB manual [5] describes it as follows:

> Displaced stepping is a way to single-step over breakpoints without removing them from the inferior, by executing an out-of-line copy of the instruction that was originally at the breakpoint location. It is also known as out-of-line single-stepping.

What Displaced Stepping does is copying the instruction to a different location within the target’s address space (scratch space), fix it if needed and execute it there. This allows the original breakpoint instruction to remain at the target address undisturbed while still letting the original instruction execute safely. After the displaced step is performed, GDB updates the CPU context (adjusting the instruction pointer) and the program continues its execution.

The function below is the implementation in GDB that handles Displaced Stepping. We even found its initial commit on the GDB source code [6]. What this function does is the following:

1. Reads the targeted instruction from memory;
2. Parses it and identifies it;
3. Checks if it’s a syscall;
4. Patches it if needed;
5. Writes it to the new location and;
6. Executes it.

The function below doesn’t execute the code, but as we will see, it is executed from the scratch space.

File: binutils-gdb/gdb/amd64-tdep.c

```
1626 displaced_step_copy_insn_closure_up
1627 amd64_displaced_step_copy_insn (struct gdbarch *gdbarch,
1628                                 CORE_ADDR from, CORE_ADDR to,
1629                                 struct regcache *regs)
1630 {
1631   int len = gdbarch_max_insn_length (gdbarch);
...
1635   std::unique_ptr<amd64_displaced_step_copy_insn_closure> dsc
1636     (new amd64_displaced_step_copy_insn_closure (len + fixup_sentinel_space));
1637   gdb_byte *buf = &dsc->insn_buf[0];
1638   struct amd64_insn *details = &dsc->insn_details;
...
1640   read_memory (from, buf, len); [1]
...
1647   amd64_get_insn_details (buf, details); [2]
...
1657     if (amd64_syscall_p (details, &syscall_length)) [3]
...
1661   /* Modify the insn to cope with the address where it will be executed from.
1662      In particular, handle any rip-relative addressing.  */
1663   if (!fixup_displaced_copy (gdbarch, dsc.get (), from, to, regs)) [4]
...
1666   write_memory (to, buf, len); [5]
...
1674 }
```

After enabling the displaced stepping debugging configuration in GDB (`set debug displaced on`), we could see the inner workings of Displaced Stepping at runtime. That's how we learned that GDB was in fact executing the instruction out-of-line at the address `0x401092` and not 0x00007fffffffe050.

[User-mode GDB]

```
[displaced] amd64_displaced_step_copy_insn: copy 0x7fffffffe050->0x401092: 00 90 90 90
```

Now, we define a hardware breakpoint at the new location in the kernel debugger, and we finally can see the corrupted shellcode being executed.

[User-mode GDB]

```
(gdb) hbreak *0x401092
Hardware assisted breakpoint 8 at 0x401092
```

```
(gdb) c
Continuing.

Thread 1 hit Breakpoint 8, 0x0000000000401092 in ?? ()
=> 0x0000000000401092:  00 90 90 90 90 90       add    %dl,-0x6f6f6f70(%rax)
(gdb)
```

If the configuration `displaced-stepping` is disabled, the breakpoint is still hit, and the shellcode is genuinely corrupted. This time, we can even see it being executed by the CPU through kernel debugging at the proper location. The shellcode ends up being corrupted because the virtual breakpoint hit makes GDB to restore the original byte. This happens although the breakpoint instruction has been overwritten by the shellcode and the configuration “set breakpoint always-inserted” is enabled.

[User-mode GDB]



```
(gdb) set displaced-stepping off
(gdb) set breakpoint always-inserted on
(gdb) r
Starting program: /home/user/work/linux-binary-exploitation-exercises-development/exer
...
buffer: 0x7fffffffe050
Hello ������������������������������������������������������������������������������������
Breakpoint 3, 0x0000000000401228 in main ()
=> 0x0000000000401228 <main+178>:       c3      ret
(gdb) c
Continuing.

Breakpoint 1, 0x00007fffffffe050 in ?? ()
=> 0x00007fffffffe050:  00 90 90 90 90 90       add    %dl,-0x6f6f6f70(%rax)
(gdb) c
Continuing.

Program received signal SIGSEGV, Segmentation fault.
0x00007fffffffe050 in ?? ()
=> 0x00007fffffffe050:  00 90 90 90 90 90       add    %dl,-0x6f6f6f70(%rax)
(gdb)
```

[Kernel Debugger — VMware]



```
(gdb) c
Continuing.
...

Thread 4 hit Breakpoint 1, 0x00007ffffffe050 in ?? ()
=> 0x00007ffffffe050:  00 90 90 90 90 90       add    %dl,-0x6f6f6f70(%rax)
(gdb) c
Continuing.
```

Conclusion

When we started writing this blog post just a few days ago, the goal was simply to use the `crash` utility and kernel debugging to expose the illusion GDB creates, and to show how important it is to understand the technical details involved even when exploiting a classic stack-based buffer overflow. However, we didn't expect to stumble upon behaviors we hadn't seen before, which ultimately forced us to learn a lot more about how GDB operates under the hood.

GDB is a tool we use every single day, as our daily routine constantly involves attaching GDB to VMware or QEMU for kernel debugging. Yet, this deep dive demonstrates exactly why having solid, fundamental skills is so critical. While a stack-based buffer overflow is a classic scenario, understanding how every piece connects together, including the abstractions GDB implements, requires an advanced knowledge of Linux internals, memory management and curiosity. The final lesson we learned, and one we will probably keep sharing, is that even the simplest, most well-worn concepts can teach us the greatest lessons if we just look closely enough.

References

[1] – Gdb toggling breakpoints in self-modifying code
https://stackoverflow.com/questions/11600541/gdb-toggling-breakpoints-in-self-modifying-code

[2] – How GDB "Stops" Time: A Deep Dive into Software Breakpoints
https://deepdives.medium.com/how-gdb-stops-time-a-deep-dive-into-software-breakpoints-1a74824d39be

[3] – Changes in GDB 14
https://sourceware.org/git/gitweb.cgi?p=binutils-gdb.git;a=blob_plain;f=gdb/NEWS;hb=gdb-14.1-release

[4] – 22.10 Optional Messages about Internal Happenings
https://sourceware.org/gdb/current/onlinedocs/gdb.html/Debugging-Output.html

[5] – Appendix D Maintenance Commands
https://sourceware.org/gdb/current/onlinedocs/gdb.html/Maintenance-Commands.html

[6] – Implement displaced stepping.
https://sourceware.org/git/?p=binutils-gdb.git;a=commit;h=237fc4c9cdd

© 2026 Allele Security Intelligence.
All rights reserved. Privacy Policy.