{binary hacking basics}

<A hands on approach for application security>

{Md.Nazmuddoha Ansary Shohag}
{B.Sc,EEE,BUET}

{__init__}

{‘the truth is rarely pure and never simple ’}
--oscar wilde

>> helpful prescience : things that will help if you know a bit
{linux terminal : basic commands}
{c,python,assembly : basic idea}

>> tools : what you need to have
{a linux os : ubuntu will be used for now}

>> topics : what we will cover
basic concepts <RPN,STACK,MEMORY>
bypass a password checking binary <BREAK IT TO HACK IT>
see ways to know the actual password <PEEKING>
reverse engineer the binary <REVERSE ENGINEERING, KEYGEN>
make the binary unhackable!!(sort of) <PARSER DIFFERENTIAL>
see how the kernel works at low level <SYSCALLS>

{RPN}

>> 2 + 5 is an infix expression: because the operator is inserted in between the two operands
>> well can we write + 2 5 to do the same thing as 2 + 5?
>> yes.. It's a prefix expression much like c function prototypes like add(x,y)⇒ add x y⇒ + x y
>> well can we write 2 5 +? Yes of course, it's called postfix
>> as a matter of fact this postfix lays at the heart of it all that happens in a computer
>> Polish mathematician 'Jan Lucasiewicz' proposed the prefix expressions because it helped him to do his mathematical proofs in a better way and later Australian philosopher and computer scientist Charles L. Hamblin suggested placing the operator after the operands (postfix)
>> say we have 2+5x3 in decimals. 2 + 5 x 3= 17 as we all know. And if we wanted to give priority to the addition we would have to write (2+5)x3 which is 21.
>> Lucasiewicz hated to use the parentheses in his proofs and wrote in prefix notations. But his name 'is' too difficult to pronounce and for simplicity people called it POLISH NOTATION. After the reversal was proposed by Hamblin, REVERSE POLISH NOTATION or RPN came to existence.
>> in RPN **2 + 5 x 3 → 2 5 3 x +** and **(2+5)x3 → 2 5 + 3 x**
>> the way it works is from left to right whenever it encounters an operator, it executes the operation on the immediately preceding two operands, to illustrate :

```
2 5 3 x + -- if found an operator
2 5 3 x + -- looks at the immediate two operands and executes
2 15 ± -- found another
2 15 ± -- looks at the immediate two operands and executes
17
2 5 ± 3 x
2 5 ± 3 x
7 3 x
7 3 x
21
```

{RPN}

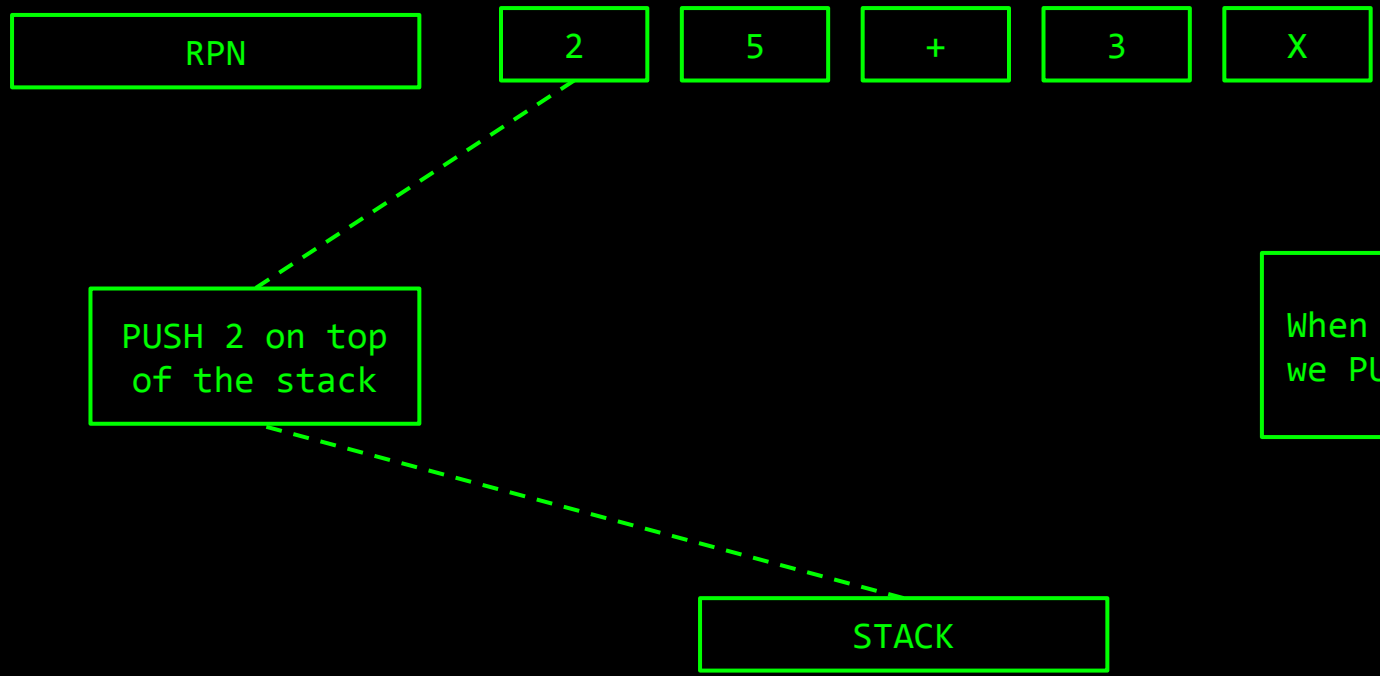
<< so what does any of it has to do with computer security??
>> well notice the add x y in Polish notation -- looks somewhat like assembly right?
>> indeed it is -- the way computers work is that they have registers and say we created an assembly language of our own (there are quite a few depending on architecture like at&t and intel) to see the result we would do something like:

```
Load R1 , x   | loads x in R1 register -- in memory
Load R2 , y   | loads y in R2 register -- in memory
Add  R1 , R2  | adds R1's and R2's value and saves in R1 -- calls the arithmetic unit inside the CPU to add
```

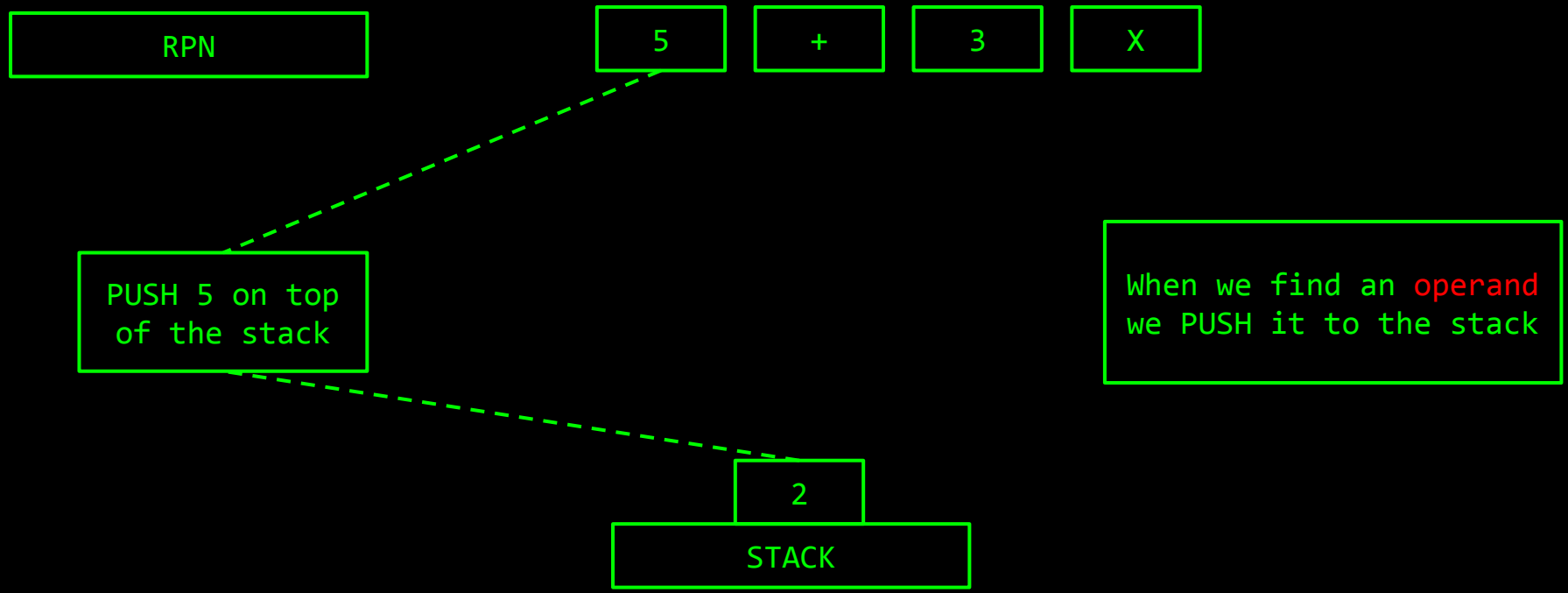
>> it's not 'strictly speaking' accurate but will suffice for simplicity
>> What about the RPN?
>> Well as it turns out using RPN actually helps compilers to save an awful lot of efforts
>> Also it is related a core concept in how the computer processes: THE STACK
>> A stack is something where you can pile things up
>> in other words you can PUSH things on it and POP things off the top
>> So it's a 'last thing in-- first thing out' storage mechanism

>> Let's give it a closer look shall we?

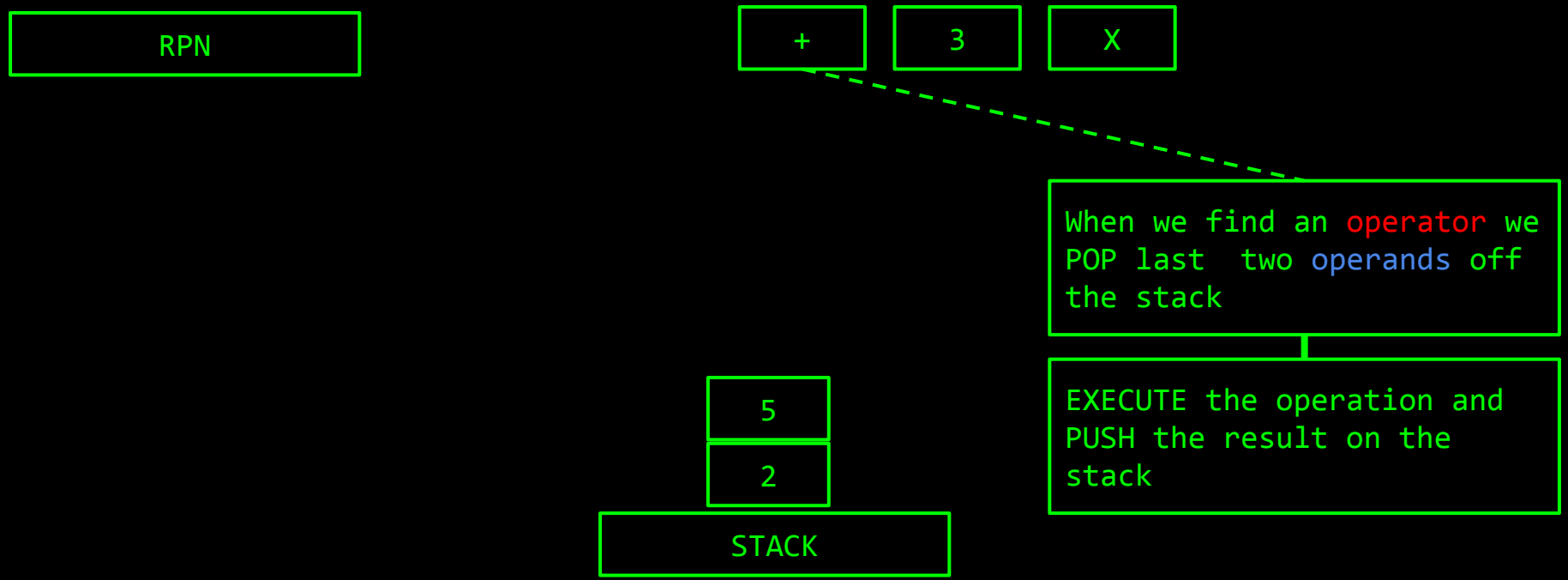
{stack}


RPN
2
5
+
3
X
PUSH 2 on top of the stack
When we find an operand we PUSH it to the stack
STACK

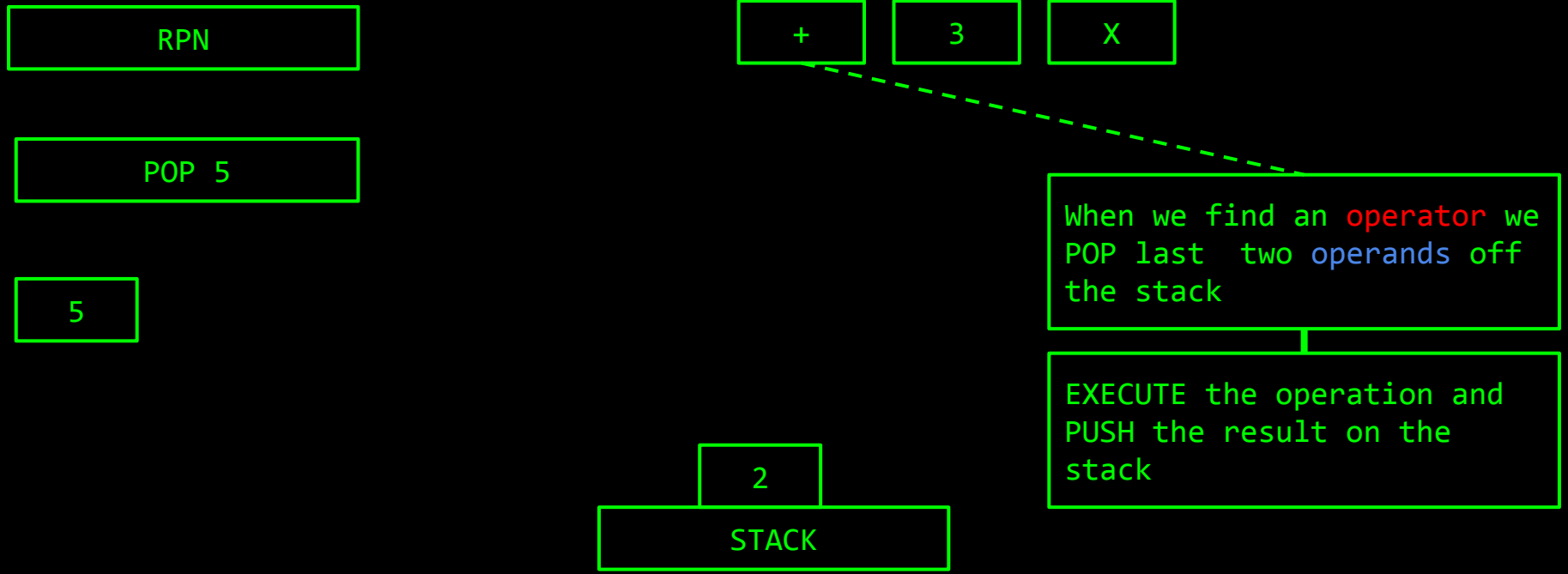
{stack}


RPN
5
+
3
X
PUSH 5 on top of the stack
When we find an operand we PUSH it to the stack
2
STACK

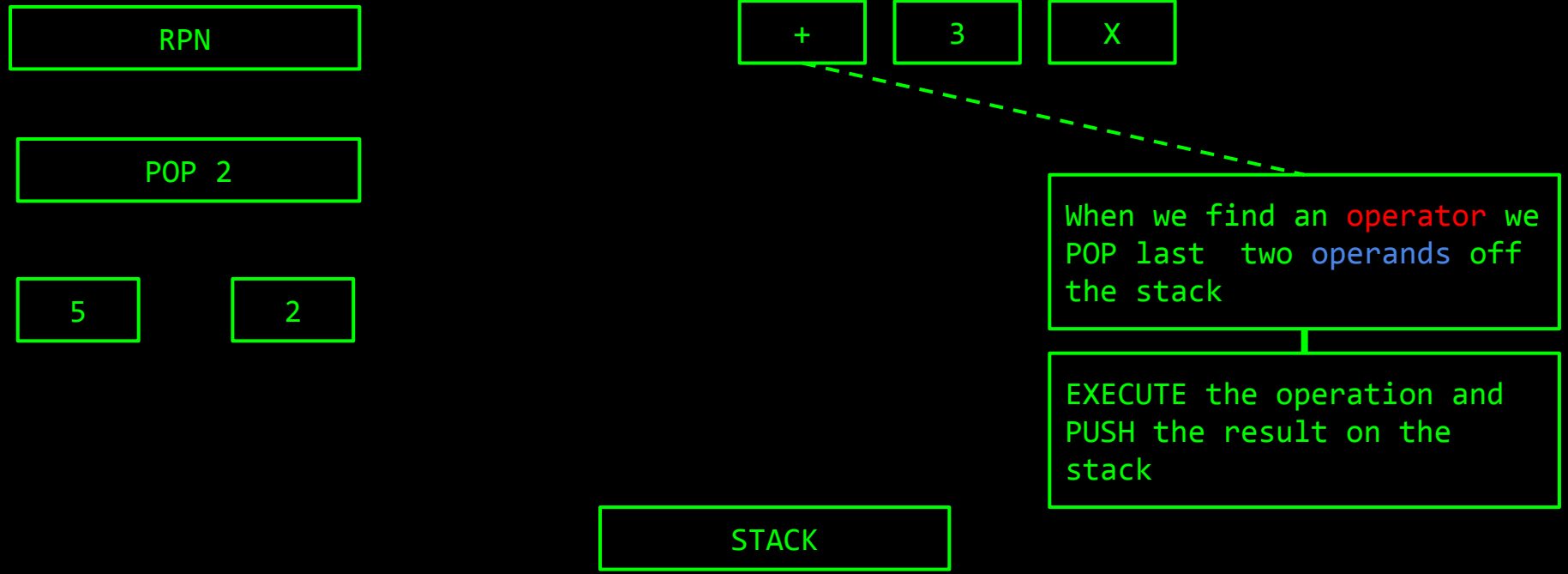
{stack}


RPN
+
3
X
When we find an operator we POP last two operands off the stack
EXECUTE the operation and PUSH the result on the stack
5
2
STACK

{stack}


RPN
POP 5
5
+
3
X
When we find an operator we POP last two operands off the stack
EXECUTE the operation and PUSH the result on the stack
2
STACK

{stack}


RPN
POP 2
5
2
+
3
X
When we find an operator we POP last two operands off the stack
EXECUTE the operation and PUSH the result on the stack
STACK

{stack}

RPN

EXEC op

5	+	2

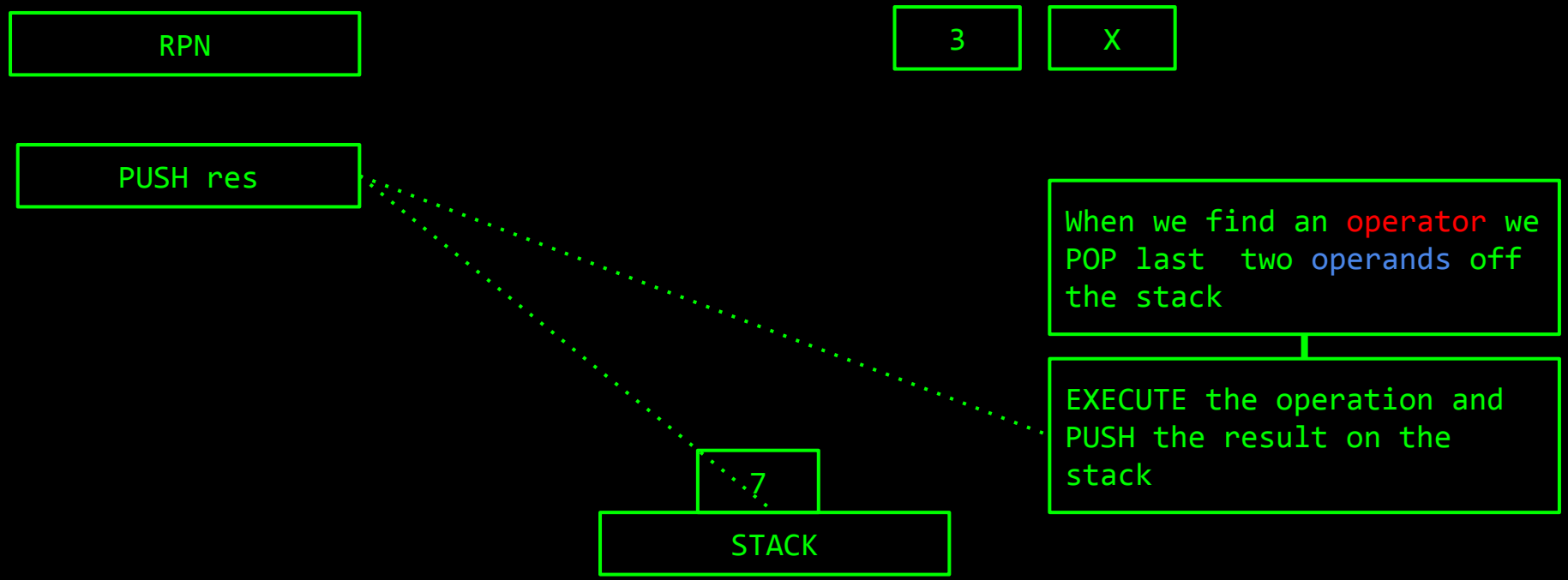
3	X

When we find an operator we POP last two operands off the stack

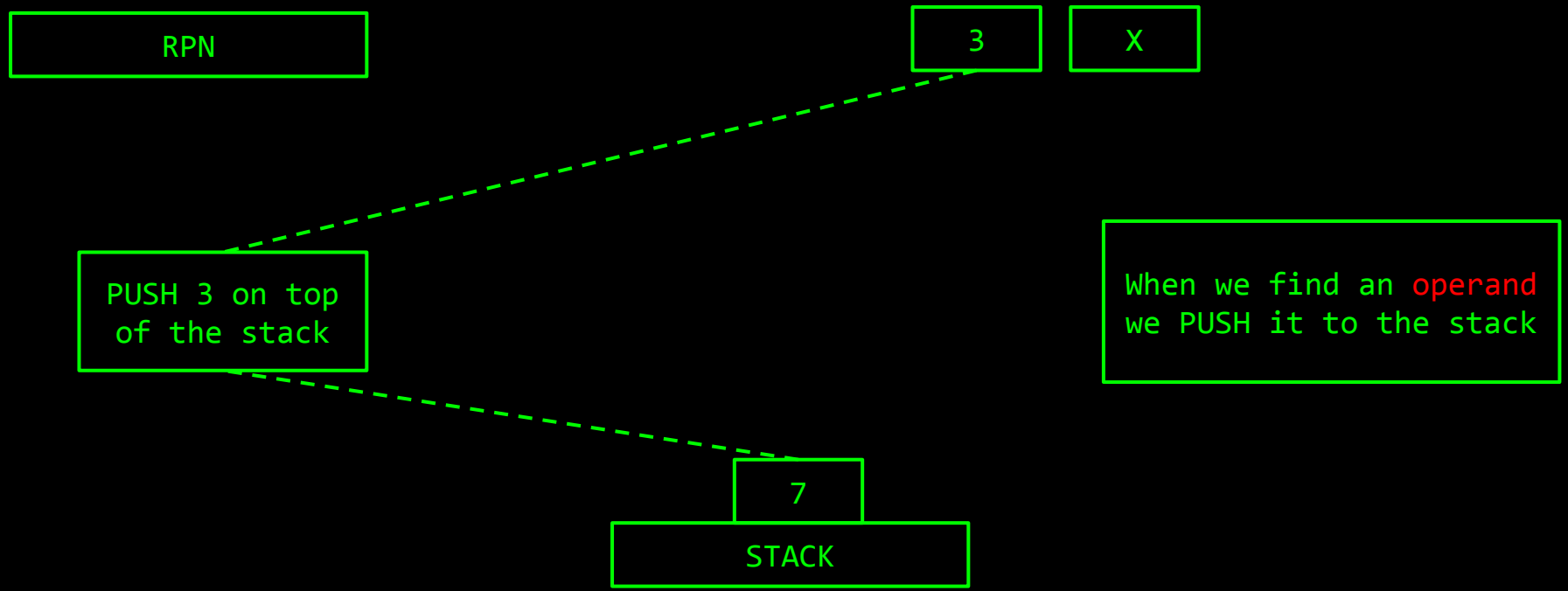
EXECUTE the operation and PUSH the result on the stack

STACK

{stack}


RPN
3
X
PUSH res
When we find an operator we POP last two operands off the stack
EXECUTE the operation and PUSH the result on the stack
7
STACK

{stack}


RPN
3
X
PUSH 3 on top
of the stack
When we find an operand
we PUSH it to the stack
7
STACK

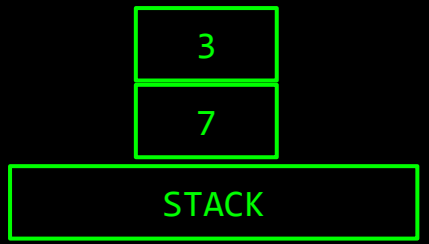
{stack}

RPN

X

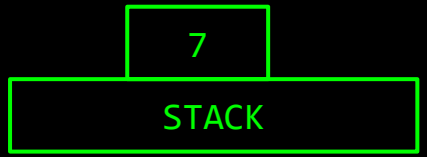
When we find an operator we POP last two operands off the stack

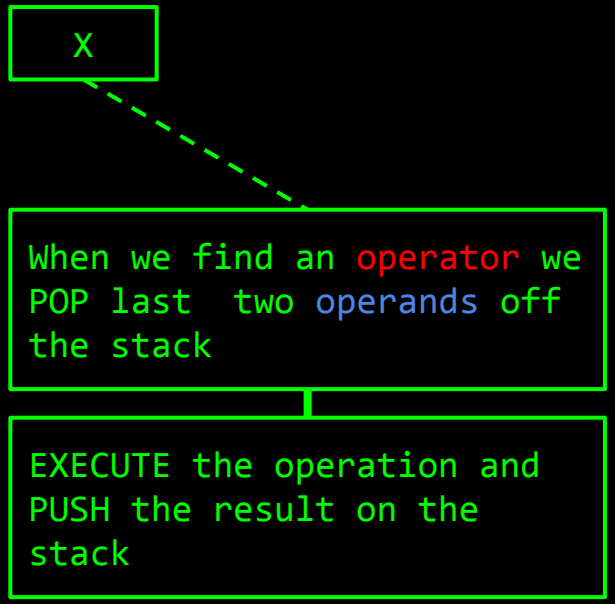
EXECUTE the operation and PUSH the result on the stack



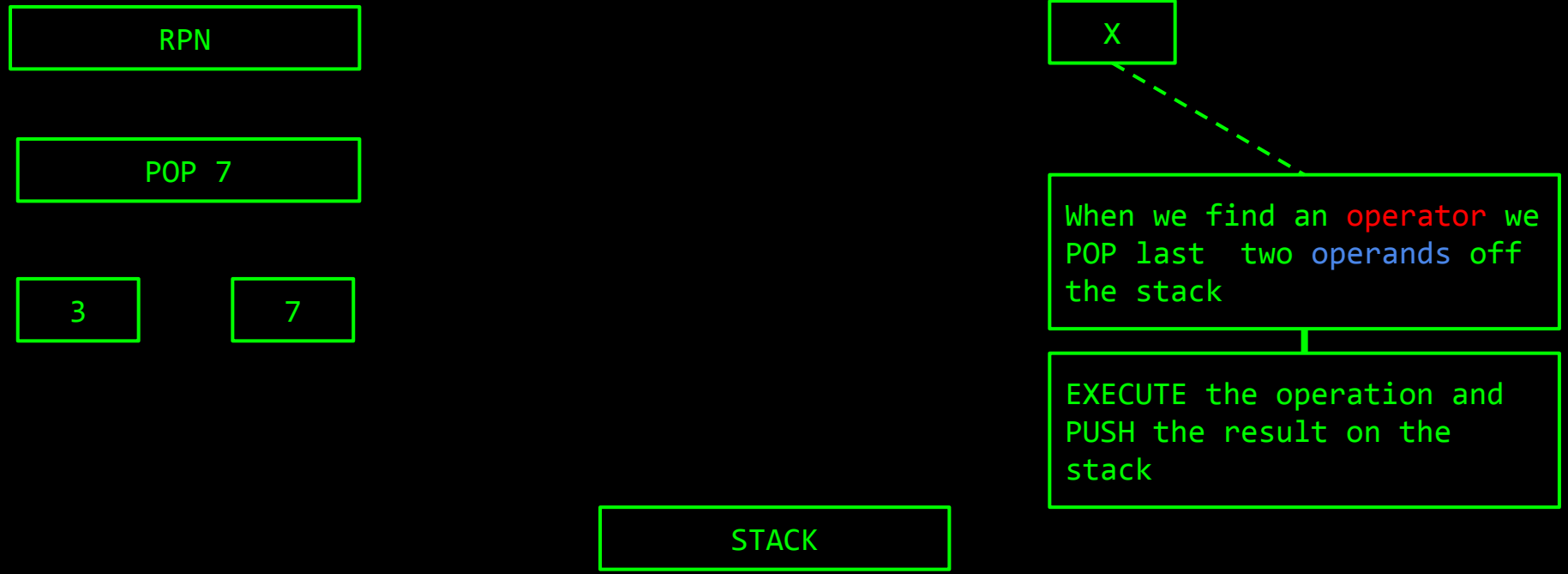
{stack}


RPN
POP 3
3


7
STACK


X
When we find an operator we POP last two operands off the stack
EXECUTE the operation and PUSH the result on the stack

{stack}

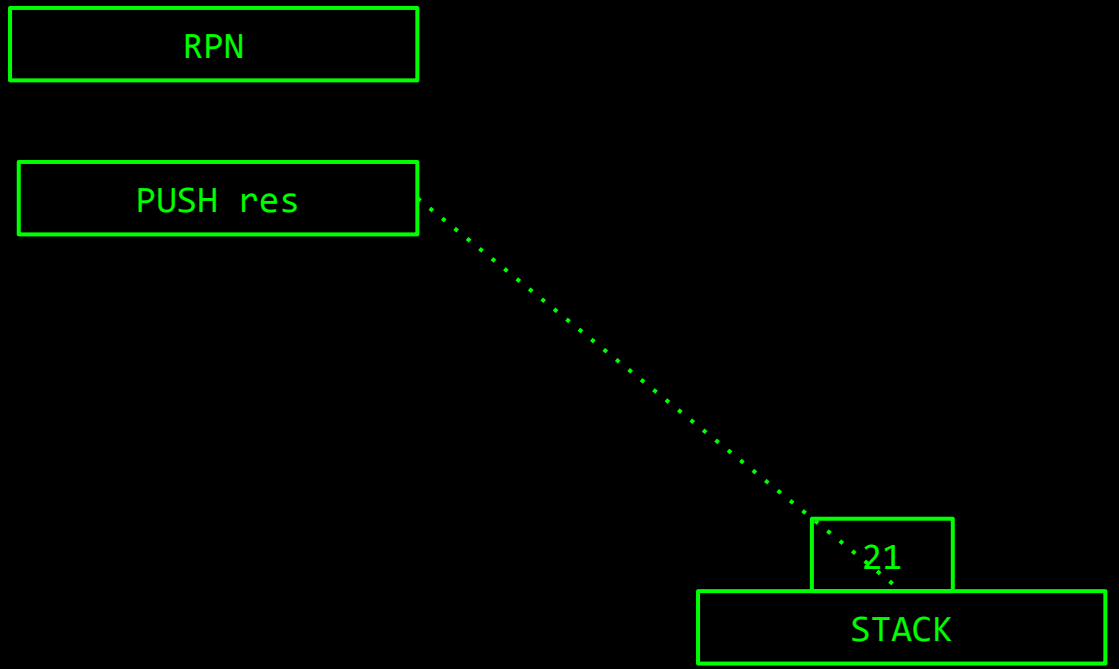

RPN
POP 7
3
7
X
When we find an operator we POP last two operands off the stack
EXECUTE the operation and PUSH the result on the stack
STACK

{stack}

When we find an operator we POP last two operands off the stack

EXECUTE the operation and PUSH the result on the stack

{stack}



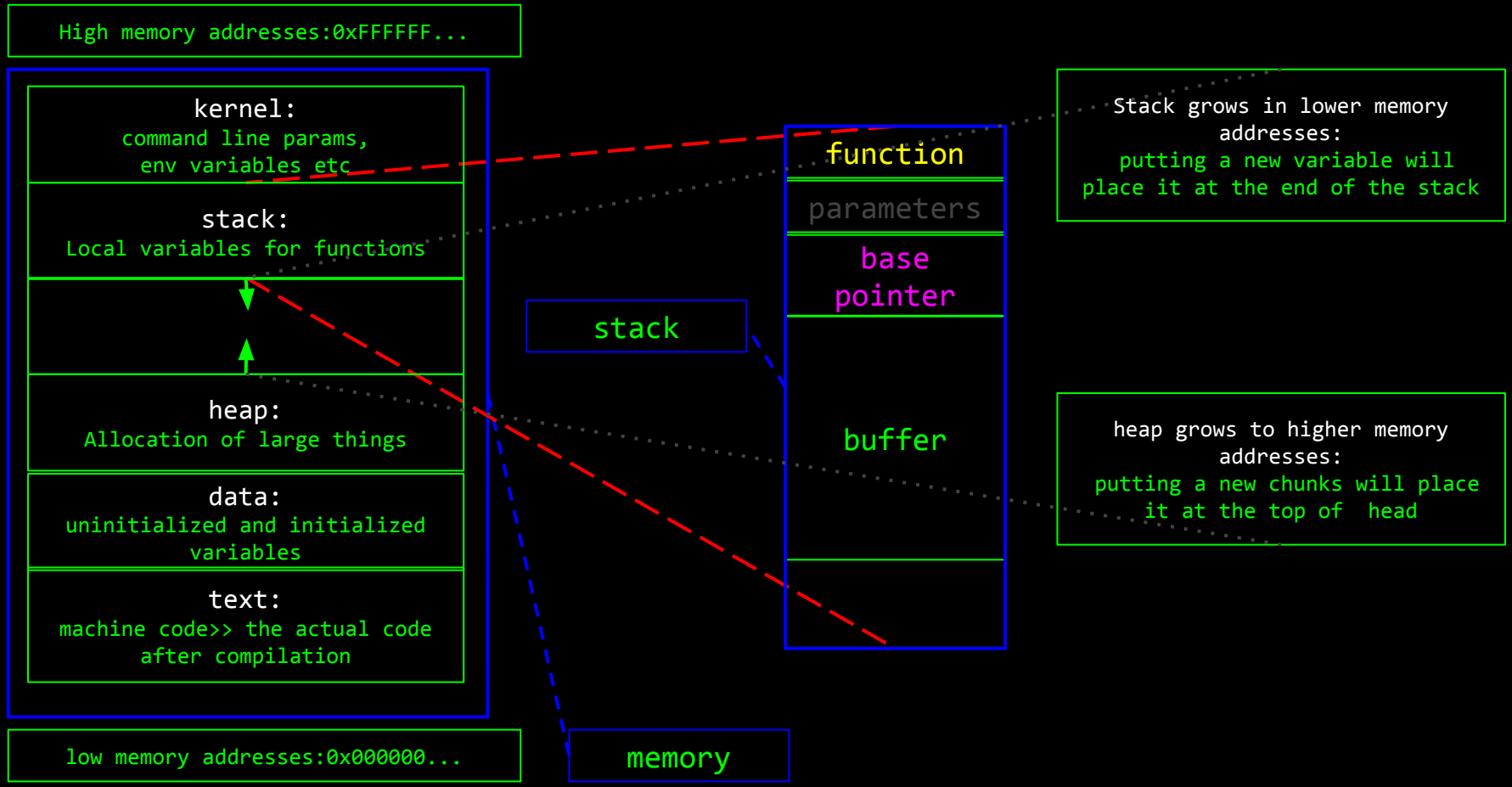
When we find an operator we POP last two operands off the stack

EXECUTE the operation and PUSH the result on the stack

{stack}

>> NOW we have an idea how the stacking takes place
>> BUT Where is the STACK ?

{memory}


High memory addresses:0xFFFFFF...
kernel:
command line params,
env variables etc
stack:
Local variables for functions
heap:
Allocation of large things
data:
uninitialized and initialized
variables
text:
machine code>> the actual code
after compilation
low memory addresses:0x000000...
stack
memory
function
parameters
base
pointer
buffer
Stack grows in lower memory
addresses:
putting a new variable will
place it at the end of the stack
heap grows to higher memory
addresses:
putting a new chunks will place
it at the top of head

{break}

>> Now as we have some basic idea about stack and how the memory is allocated:
brush up on your assembly a little bit (if needed)

Fire up your linux machine and lets get into hacking a binary

{ code it to know it }

```
# sudo apt install  $THINGS {THINGS : vim/gcc/gdb etc}
>> type in terminal $ vim binary101.c
>> vim_basic_commands
# {press i to enter insert mode}
# {press esc to exit insert mode}
>> while not in insert mode:
       print('press :q to exit vim')
       print('press :wq to save and exit vim')
       If (fancy_layout==True):
              print('type :syntax on')
              print('type :set number')
>> if (layout_still_dissapointing==True):
      print('THAT IS THE BEST YOU GET FOR NOW!')
# write the code
# compile the c source into a binary
$ gcc binary101.c  -o testbinary -Wall
# check the binary
$ ./testbinary
Should print: Usage: ./testbinary  <key>

$ ./testbinary anyRandomKey
Should print: Checking pass key: anyRandomKey
              WRONG!

$./testbinary  THE_SECRECT_KEY
Should print: Checking pass key: THE_SECRECT_KEY
              Access Granted!
    (notice the 'C' in secreCt?-- it's intentional)
```

ansary@ansary-System-Product-Name: ~/BINARY

File Edit View Search Terminal Help

```
1  //simple c code that checks a pass key
2
3  #include <stdio.h>
4  #include <string.h>
5
6  int main(int argc, char *argv[])
7  {
8      if(argc==2)
9      {
10         printf("Checking pass key: %s\n", argv[1]);
11
12         if(strcmp(argv[1], "THE_SECRECT_KEY")==0)
13         {
14                 printf("Access Granted!\n");
15         }
16         else
17         {
18                 printf("WRONG!\n");
19         }
20         return 0;
21     }
22     else
23     {
24                 fprintf(stderr,"Usage: %s  <key>\n",argv[0]);
25                 return 1;
26     }
27 }
```

1,26 All

My crime is that of curiosity

{ break it to hack it }

```
$ gdb testbinary                    | for opening the binary to view the assembly code
(gdb) set disassembly-flavor intel  | because we don't want at&t syntax to ruin our day
(gdb) disassemble main              | disassemble main function
```

```
(gdb) set disassembly-flavor intel
(gdb) disassemble main
Dump of assembler code for function main:
   0x000000000000075a <+0>:     push   rbp
   0x000000000000075b <+1>:     mov    rbp,rsp
   0x000000000000075e <+4>:     sub    rsp,0x10
   0x0000000000000762 <+8>:     mov    DWORD PTR [rbp-0x4],edi
   0x0000000000000765 <+11>:    mov    QWORD PTR [rbp-0x10],rsi
   0x0000000000000769 <+15>:    cmp    DWORD PTR [rbp-0x4],0x2
   0x000000000000076d <+19>:    jne    0x7cd <main+115>
   0x000000000000076f <+21>:    mov    rax,QWORD PTR [rbp-0x10]
   0x0000000000000773 <+25>:    add    rax,0x8
   0x0000000000000777 <+29>:    mov    rax,QWORD PTR [rax]
   0x000000000000077a <+32>:    mov    rsi,rax
   0x000000000000077d <+35>:    lea    rdi,[rip+0x100]        # 0x884
   0x0000000000000784 <+42>:    mov    eax,0x0
   0x0000000000000789 <+47>:    call   0x610 <printf@plt>
   0x000000000000078e <+52>:    mov    rax,QWORD PTR [rbp-0x10]
   0x0000000000000792 <+56>:    add    rax,0x8
   0x0000000000000796 <+60>:    mov    rax,QWORD PTR [rax]
   0x0000000000000799 <+63>:    lea    rsi,[rip+0xfb]        # 0x89b
   0x00000000000007a0 <+70>:    mov    rdi,rax
   0x00000000000007a3 <+73>:    call   0x620 <strcmp@plt>
   0x00000000000007a8 <+78>:    test   eax,eax
   0x00000000000007aa <+80>:    jne    0x7ba <main+96>
   0x00000000000007ac <+82>:    lea    rdi,[rip+0xf8]        # 0x8ab
   0x00000000000007b3 <+89>:    call   0x600 <puts@plt>
   0x00000000000007b8 <+94>:    jmp    0x7c6 <main+108>
   0x00000000000007ba <+96>:    lea    rdi,[rip+0xfa]        # 0x8bb
---Type <return> to continue, or q <return> to quit---
   0x00000000000007c1 <+103>:   call   0x600 <puts@plt>
   0x00000000000007c6 <+108>:   mov    eax,0x0
   0x00000000000007cb <+113>:   jmp    0x7f4 <main+154>
   0x00000000000007cd <+115>:   mov    rax,QWORD PTR [rbp-0x10]
   0x00000000000007d1 <+119>:   mov    rdx,QWORD PTR [rax]
   0x00000000000007d4 <+122>:   mov    rax,QWORD PTR [rip+0x200845]        # 0x201020 <stderr@@GLIBC_2.2.5>
   0x00000000000007db <+129>:   lea    rsi,[rip+0xe0]        # 0x8c2
   0x00000000000007e2 <+136>:   mov    rdi,rax
   0x00000000000007e5 <+139>:   mov    eax,0x0
   0x00000000000007ea <+144>:   call   0x630 <fprintf@plt>
   0x00000000000007ef <+149>:   mov    eax,0x1
   0x00000000000007f4 <+154>:   leave
   0x00000000000007f5 <+155>:   ret
End of assembler dump.
(gdb)
```

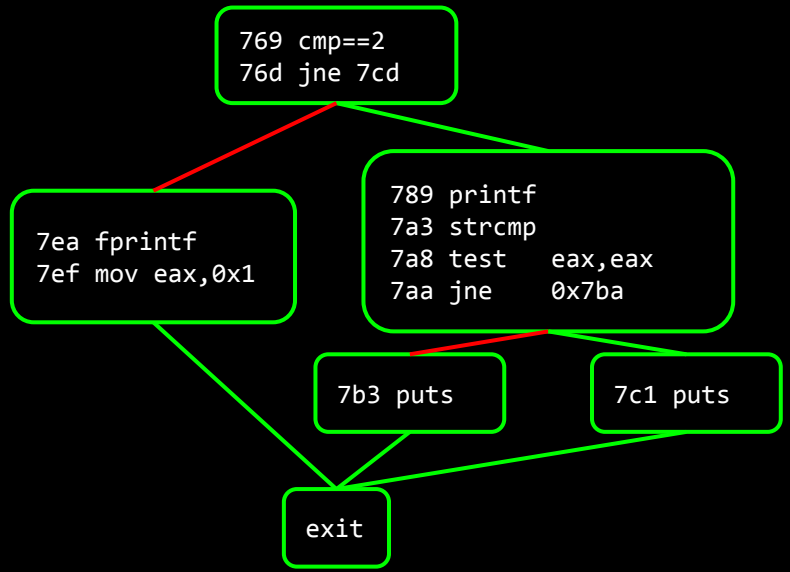
```
>>Control_flow (consider only the essentials)
769 <+15>:    cmp    DWORD PTR [rbp-0x4],0x2
76d <+19>:    jne    0x7cd <main+115>
{near the start a compare is done with number 2}
7cd <+115>:   mov    rax,QWORD PTR [rbp-0x10]
7ea <+144>:   call   0x630 <fprintf@plt>
7ef <+149>:   mov    eax,0x1
{If not equal: calls fprintf and exits with 0x1 } else:
789 <+47>:    call   0x610 <printf@plt>
7a3 <+73>:    call   0x620 <strcmp@plt>
7a8 <+78>:    test   eax,eax
7aa <+80>:    jne    0x7ba <main+96>
{now some string is compared}

# If equal: prints something and exits
7b3 <+89>:    call   0x600 <puts@plt>
7b8 <+94>:    jmp    0x7c6 <main+108>
# Else: prints something else and exits
7c1 <+103>:   call   0x600 <puts@plt>
```

<Note that 0X7c6 starts the flow for leave and return in both cases>

{ control flow (contd.) }

```
#let's check the control flow under different conditions
(gdb) break *main                | sets break point
(gdb) run                        | equivalent to ./testbinary
(gdb) ni                         | point to next instruction
<press enter after first ni no need for repeated typing>
{In addition we can (gdb) info registers
To check the values of the registers}
```

<Simplified control flow>




```
(gdb) run
Starting program: /home/ansary/BINARY/testbinary

Breakpoint 1, 0x000055555555475a in main ()
(gdb) ni
0x000055555555475b in main ()
(gdb)
0x000055555555475e in main ()
(gdb)
0x0000555555554762 in main ()
(gdb)
0x0000555555554765 in main ()
(gdb)
0x0000555555554769 in main ()
(gdb)
0x000055555555476d in main ()
(gdb)
0x00005555555547cd in main ()
(gdb)
0x00005555555547d1 in main ()
(gdb)
0x00005555555547d4 in main ()
(gdb)
0x00005555555547db in main ()
(gdb)
0x00005555555547e2 in main ()
(gdb)
0x00005555555547e5 in main ()
(gdb)
0x00005555555547ea in main ()
(gdb)
Usage: /home/ansary/BINARY/testbinary  <key>
0x00005555555547ef in main ()
(gdb)
0x00005555555547f4 in main ()
(gdb)
0x00005555555547f5 in main ()
(gdb)
__libc_start_main (main=0x555555555475a <main>, argc=1, argv=0x7fffffffdec8, init=<optimized
out>, fini=<optimized out>, rtld_fini=<optimized out>, stack_end=0x7fffffffdeb8) at ../csu/l
ibc-start.c:344
344     ../csu/libc-start.c: No such file or directory.
(gdb)
0x00007ffff7a05b99      344     in ../csu/libc-start.c
(gdb)
[Inferior 1 (process 9642) exited with code 01]
(gdb)
The program is not being run.
(gdb)
```

{ control flow (contd.) }

```
(gdb) run  randomkey    |./testbinary randomkey

As we go along the way we would see--
0x0000555555554784 in main ()
(gdb)
0x0000555555554789 in main ()
(gdb)
Checking pass key: random_key
Then--
0x00005555555547c1 in main ()
(gdb)
WRONG!
0x00005555555547c6 in main ()
(gdb)
0x00005555555547cb in main ()
```

```
Now if we run with the actual key we know--
(gdb) run  THE_SECRECT_KEY
.....
Checking pass key: THE_SECRECT_KEY
0x000055555555478e in main ()
(gdb)
.....
0x00005555555547ac in main ()
(gdb)
0x00005555555547b3 in main ()
(gdb)
Access Granted!
```

Observations and measurements

>> The code runs as expected without any error

>> The complete program runs as per the control flow as it should

Now Lets Hack Into This Binary

Assumptions

>> Let’s assume we don’t have any idea about the source code and the actual password

>> We want to gain access with any random string and not know the password even after exploitation

{ exploitation }

```
# { Since we don’t know the pass key now as per our assumption, we need to go back to control flow }
>> find_point_of_exploitation
7a3 <+73>:   call   0x620 <strcmp@plt>
7a8 <+78>:   test   eax,eax
7aa <+80>:   jne    0x7ba <main+96>
# { Let's inspect this section}
>> a string is being compared
>> some value is going being set with  test   eax,eax
>> ok so let’s set a breakpoint and see the registers specially eax

<note: eax is actually the first 32 bit of rax register since the system is 64 bit>


 (gdb) break *0x00005555555547a8            | sets the break point on 7a8
 Breakpoint 2 at 0x5555555547a8
 (gdb) run anyrandomthing                   | runs the program with a random string
 Starting program: /home/ansary/BINARY/testbinary anyrandomthing

 Breakpoint 1, 0x000055555555475a in main ()
 (gdb) continue                             | simply continues to next break point
 Continuing.
 Checking pass key: anyrandomthing

 Breakpoint 2, 0x00005555555547a8 in main ()
```

{ exploitation (contd.) }

```
Checking pass key: anyrandomthing
Breakpoint 2, 0x00005555555547a8 in main ()
(gdb) info registers       | shows the value of registers
rax            0xd  13
rbx            0x0  0
rcx            0x0  0
rdx            0x54 84
rsi            0x55555555489b    93824992233627
.....................................................................
>> here we see that value of rax is non-zero
>> which means the compare did not return zero (i.e- our key is wrong)
# {so let's set this to zero}

 (gdb) set $eax=0          | sets eax==0
 (gdb) info registers      | confirmation of eax being 0
 rax            0x0  0
 ..............................
 (gdb) ni                  | going to next rip i.e- instruction pointer
 0x00005555555547aa in main ()
 (gdb)
 0x00005555555547ac in main ()
 (gdb)
 0x00005555555547b3 in main ()
 (gdb)
 Access Granted!  <<<<<<<<<<<< We just got access without knowing the password!!!!
 >> type q and exit gdb
```

{ peeking }

>> Do a hexdump of the binary to find the key $ hexdump -C testbinary
>> Filter readable strings and try them as key $ strings testbinary
>> Do an objdump with headers and pipe them into less $ objdump -x testbinary | less
Heads up
{viewing at objdump we see that the stack is not executable by the missing x permission in flags}
STACK off   0x0000000000000000 vaddr 0x0000000000000000 paddr 0x0000000000000000 align 2**4
         filesz 0x0000000000000000 memsz 0x0000000000000000 flags rw-
|^| future discussion {classic buffer overflow}
also check the read only data sections
15 .rodata   00000054 0000000000000880 0000000000000880 00000880 2**2
                 CONTENTS, ALLOC, LOAD, READONLY, DATA

Task-1

>> open up gdb and disassemble main and set a breakpoint before string compare
>> check the registers -- one of the registers will hold an address from the .rodata data section
>> print the actual key by (gdb) x/s <the register value>

Task-2

>> use strace and see how the execve tells the linux kernel to execute the function and detect the write function (present in syscalls by default in linux) and works as a wrapper for printf and puts
{ignore complicated stuffs for now}
>> use ltrace and see how library function from libc is called to execute printf

{ radare2 }

radare2 >> disassembler

Follow the following:

>>installing radare2

```
$ sudo apt install git
$ git clone https://github.com/radare/radare2.git
$ cd radare2/
$ [sudo] ./sys/install.sh
```

>>using radare2

```
$ r2 testbinary
 -- You have been designated for disassembly
[0x00000650]>
```

if you remember from objdump the text section started with this address
type in ? and hit enter and you will see the help text {example: a?} -- (type==write + hit Enter)

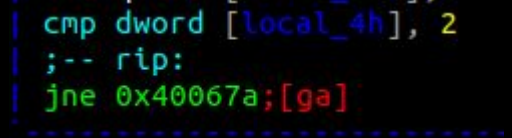
>> type aaa to automatically analyze and autoname functions
>> type afl to print all functions found
>> type s sym.main to locate the main function : see the change in location [0x0000075a]>
>> type pdf to print the disassembly of the current function : in this case main
>> type VV (capital v v not a W) to enter visual mode : to see a control graph
>> use arrows to move it around : blue box show selected part use tab and shift+tab to select different blocks
>> type p to see different representations
>> type ? to see help : see the most important shortcut you will ever use in radare???
 R - randomize colors >> press shift+R and see the magic :)

{ getting the hang of it }

TASK (20 min)

>>use file command to know the type of testbinary file produced in the previous lesson
If(testbinary: ELF 64-bit LSB shared object)
   $gcc binary101.c -no-pie -o testbinary
Make sure --testbinary: ELF 64-bit LSB executable

>>use radare2 with a -d flag
$ r2 -d testbinary
>> locate the main function
>> automatically analyze all
>> print the disassembly of the current function : in this case main
now place a breakpoint at the start of the function (use db <address>)
[0x00400607]> db 0x00400607
>> run the program with dc (with or without parameters--your choice)
>> enter visual mode --Control graph mode
>> press shift+s to step through and see if the program follows the previous described control graph
HINT: observe the rip



Experiments:
>> use V! Instead of VV and see what happens in visual mode
>> use ‘:’ to enter command mode (just like vim) and try to execute the binary

{ not having a key }

```
# {Modify the original code}
>> compare sum_of_key
# {To find original key length use python in terminal}
>> import numpy as np
>> np.fromstring("THE_SECRECT_KEY",dtype=np.uint8).sum()
1169
# BUT We will not use this value Instead we will print
the sum in our c code and use that value
$ ./testbinary THE_SERECT_KEY
Checking pass key: THE_SERECT_KEY
1102
Access Granted!
# WHY THE DIFFERENCE???? <HOME TASK>
>> now if you tried with strings or any other previous
methods you will not be able to find the key
# {OK, let's crack this shall we}
>> open this with radare2
>> analyze all(aaa)
>> seek the main function(s sym.main)
>> see the disassembly(pdf)
>> look for the Access Granted or WRONG msg
>> locate the branch that is responsible for the check
```

```c
//simple c code that checks a pass key
#include <stdio.h>
#include <string.h>
int main(int argc, char *argv[]) {
  int sum=0;
  int i;
  if(argc==2) {
     printf("Checking pass key: %s\n", argv[1]);
      for (i=0;i<strlen(argv[1]);i++){sum+=
(int)argv[1][i];}
      //printf("%d \n",sum);
     if(sum==1102) {printf("Access Granted!\n");}
      else {printf("WRONG!\n");}
      return 0;}
  else {fprintf(stderr,"Usage: %s  <key>\n",argv[0]);
      return 1;}
}
```

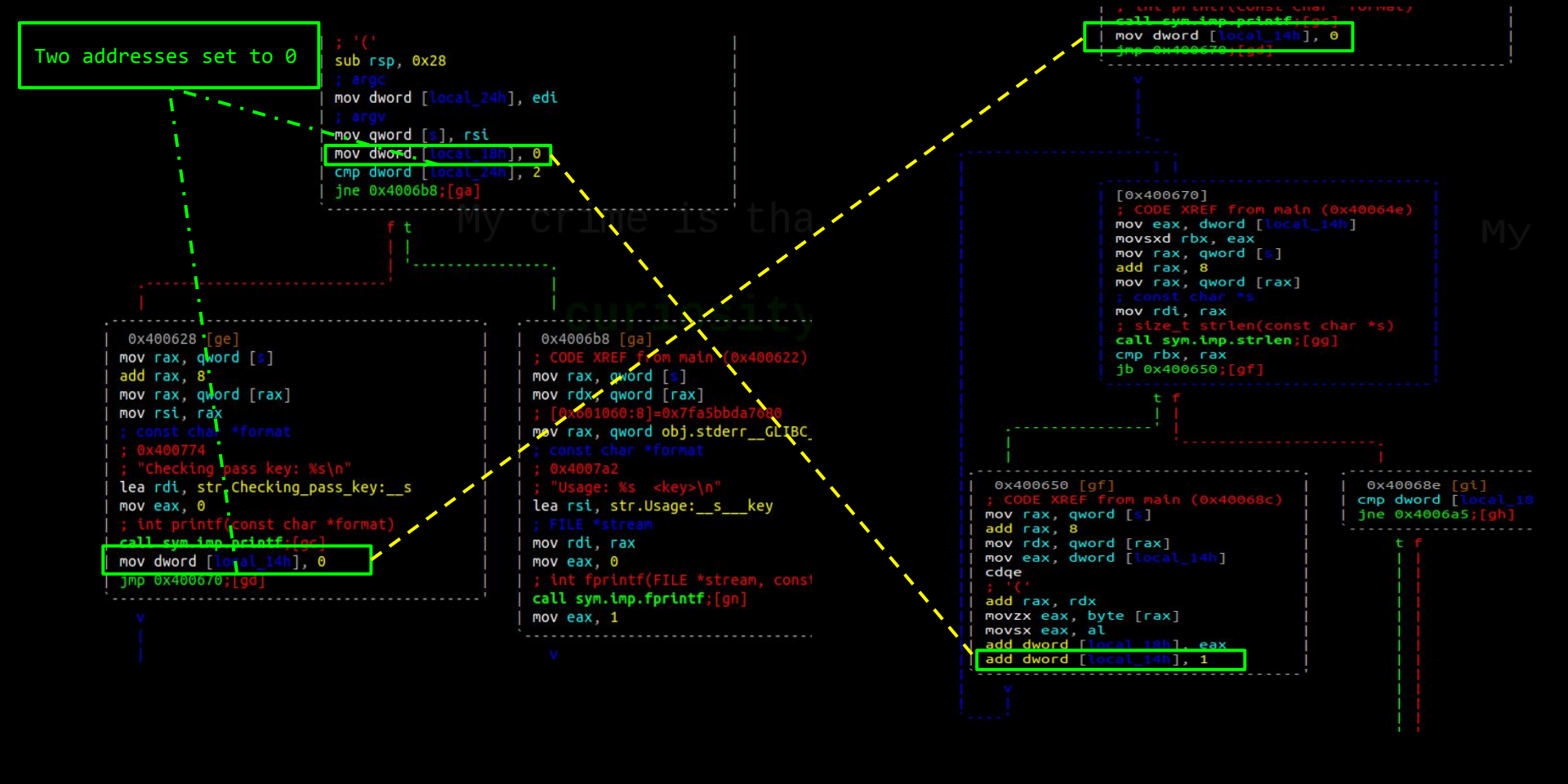
{ not having a key (contd.) }

```
|       |   0x0040069f      e85cfeffff     call sym.imp.printf         ; int printf(const char *format)
|       |   0x004006a4      817de84e0400.  cmp dword [local_18h], 0x44e
|       |   0x004006ab      750e           jne 0x4006bb
|       |   0x004006ad      488d3dec0000.  lea rdi, str.Access_Granted ; 0x4007a0 ; "Access Granted!" ; const char *s
|       |   0x004006b4      e827feffff     call sym.imp.puts           ; int puts(const char *s)
|       |   0x004006b9      eb0c           jmp 0x4006c7
|       |   ; CODE XREF from main (0x4006ab)
|       |   0x004006bb      488d3dee0000.  lea rdi, str.WRONG          ; 0x4007b0 ; "WRONG!" ; const char *s
|       |   0x004006c2      e819feffff     call sym.imp.puts           ; int puts(const char *s)
|       |   ; CODE XREF from main (0x4006b9)
|       |   0x004006c7      b800000000     mov eax, 0
|       |   0x004006cc      eb27           jmp 0x4006f5
|       |   ; CODE XREF from main (0x400622)
|       |   0x004006ce      488b45d0       mov rax, qword [s]
|       |   0x004006d2      488b10         mov rdx, qword [rax]
|       |   0x004006d5      488b05840920.  mov rax, qword [obj.stderr__GLIBC_2.2.5] ; [0x601060:8]=0
|       |   0x004006dc      488d35d40000.  lea rsi, str.Usage:__s___key ; 0x4007b7 ; "Usage: %s  <key>\n" ; const char *format
|       |   0x004006e3      4889c7         mov rdi, rax                ; FILE *stream
|       |   0x004006e6      b800000000     mov eax, 0
|       |   0x004006eb      e820feffff     call sym.imp.fprintf        ; int fprintf(FILE *stream, const char *format,   ...)
|       |   0x004006f0      b801000000     mov eax, 1
|       |   ; CODE XREF from main (0x4006cc)
|       |   0x004006f5      4883c428       add rsp, 0x28               ; '('
|           0x004006f9      5b             pop rbx
|           0x004006fa      5d             pop rbp
\           0x004006fb      c3             ret
[0x00400607]>
```

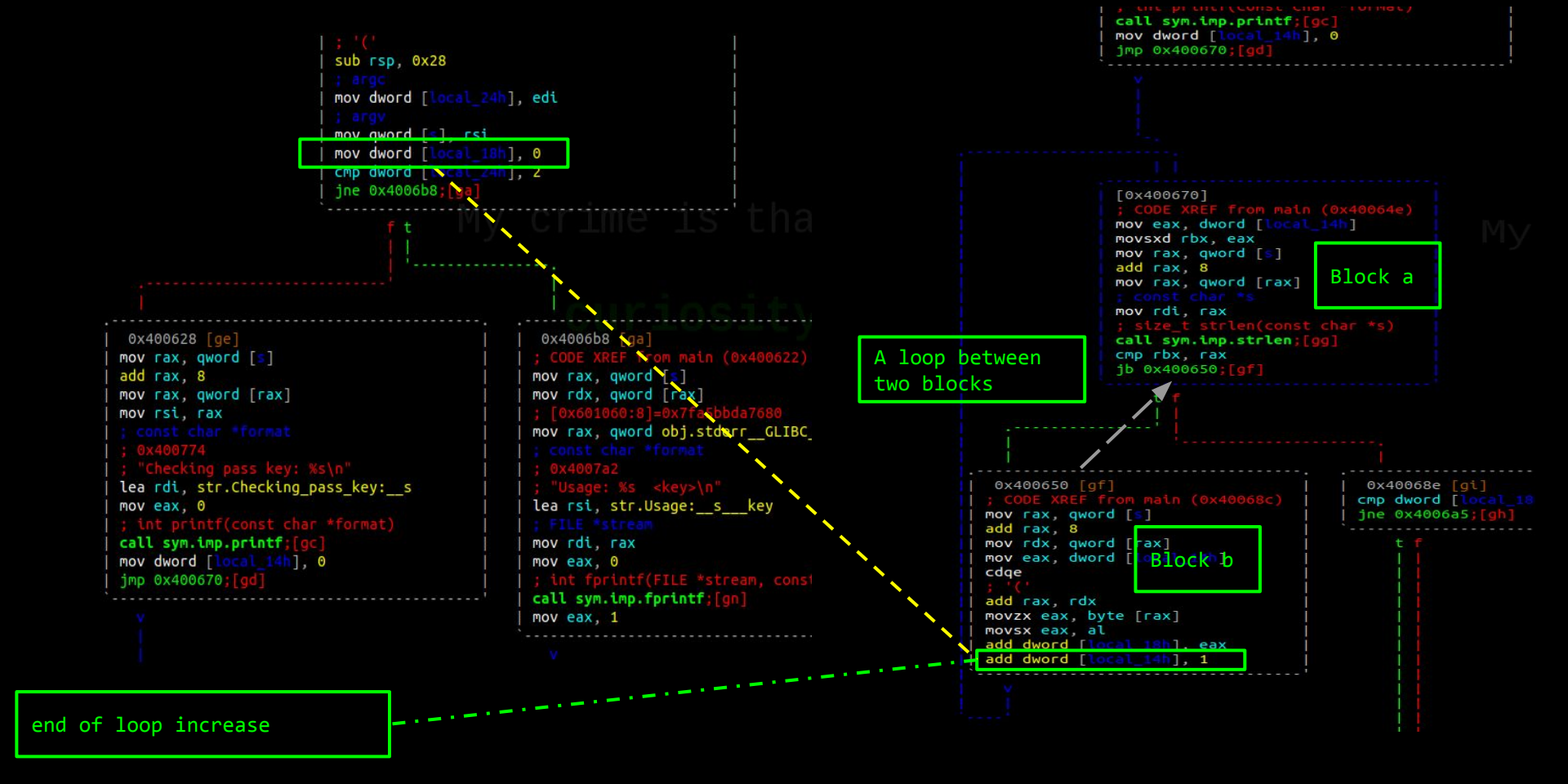
{ not having a key (contd.) }

```
>> open the file in debug mode and add a wrong key:
[0x00400607]> ood anyrandomkey
>> set a breakpoint at the compare and reopen in debug mode :
[0x7fb2b0a22e06]> db 0x00400695
[0x7fb2b0a22e06]> ood anykey
>> dc to continue to hit breakpoint and dr to view register values:
[0x7fa5bbdad090]> dc
Checking pass key: anykey
hit breakpoint at: 400695
[0x00400695]> dr
.............................
rdi = 0x7fff465cf1dd
rsp = 0x7fff465cdf20
rbp = 0x7fff465cdf50
rip = 0x00400695
rflags = 0x00000293
orax = 0xffffffffffffffff
>> set the instruction pointer to Access Granted:
[0x00400695]> dr rip=0x00400697
0x00400695 ->0x00400697
[0x00400695]> dc
Access Granted!      <<<<<<<<<< cracked it again without knowing the key
[0x7fa5bba9fe06]>
>> Although we have cracked it we don't know the key!!
>> Reverse Engineer The Binary
>> Create A keygen
>> seek to the main function and enter visual mode(aaa,s sym.main,VV)
```

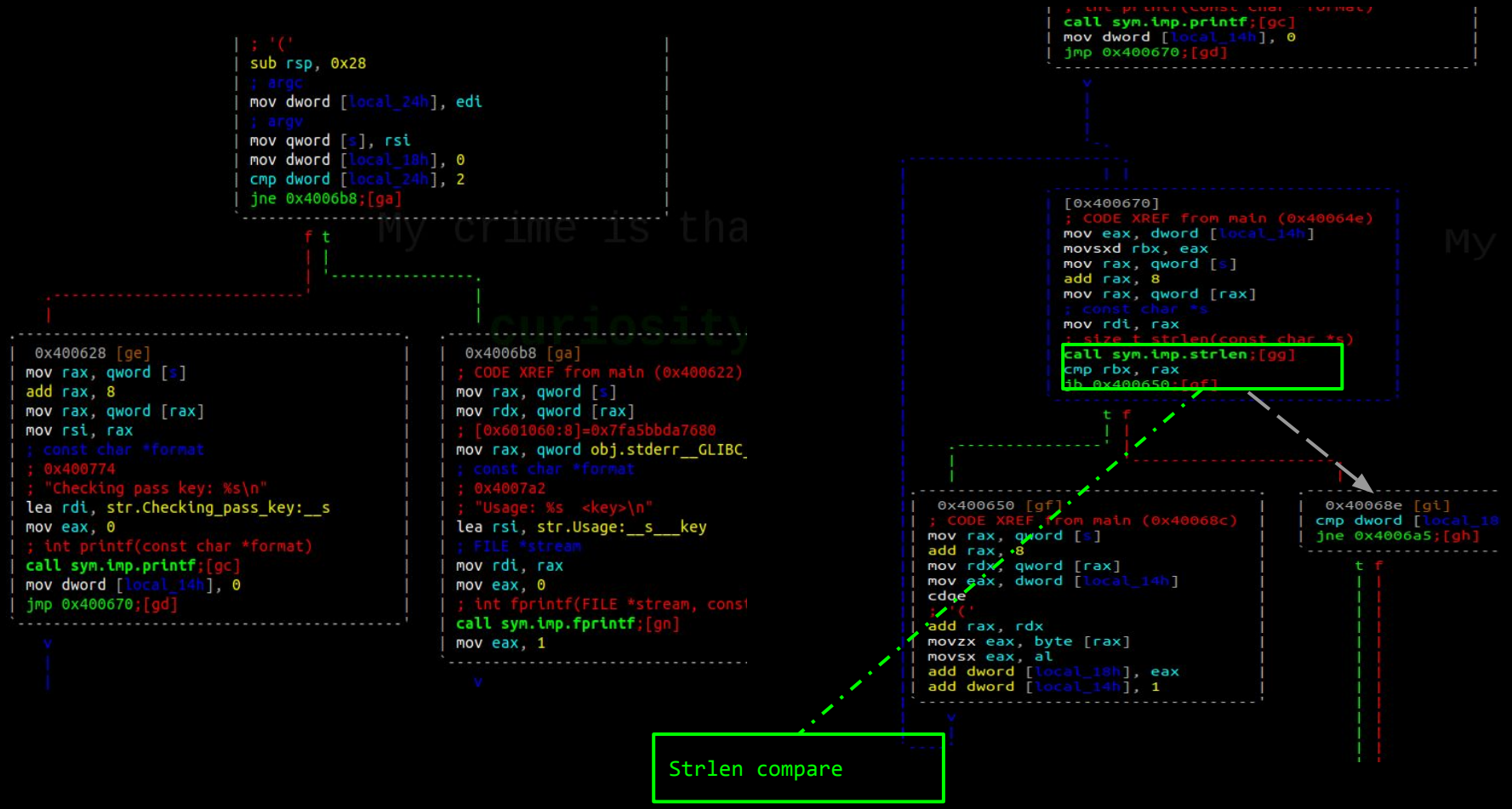
{ reverse engineering }



{ reverse engineering }

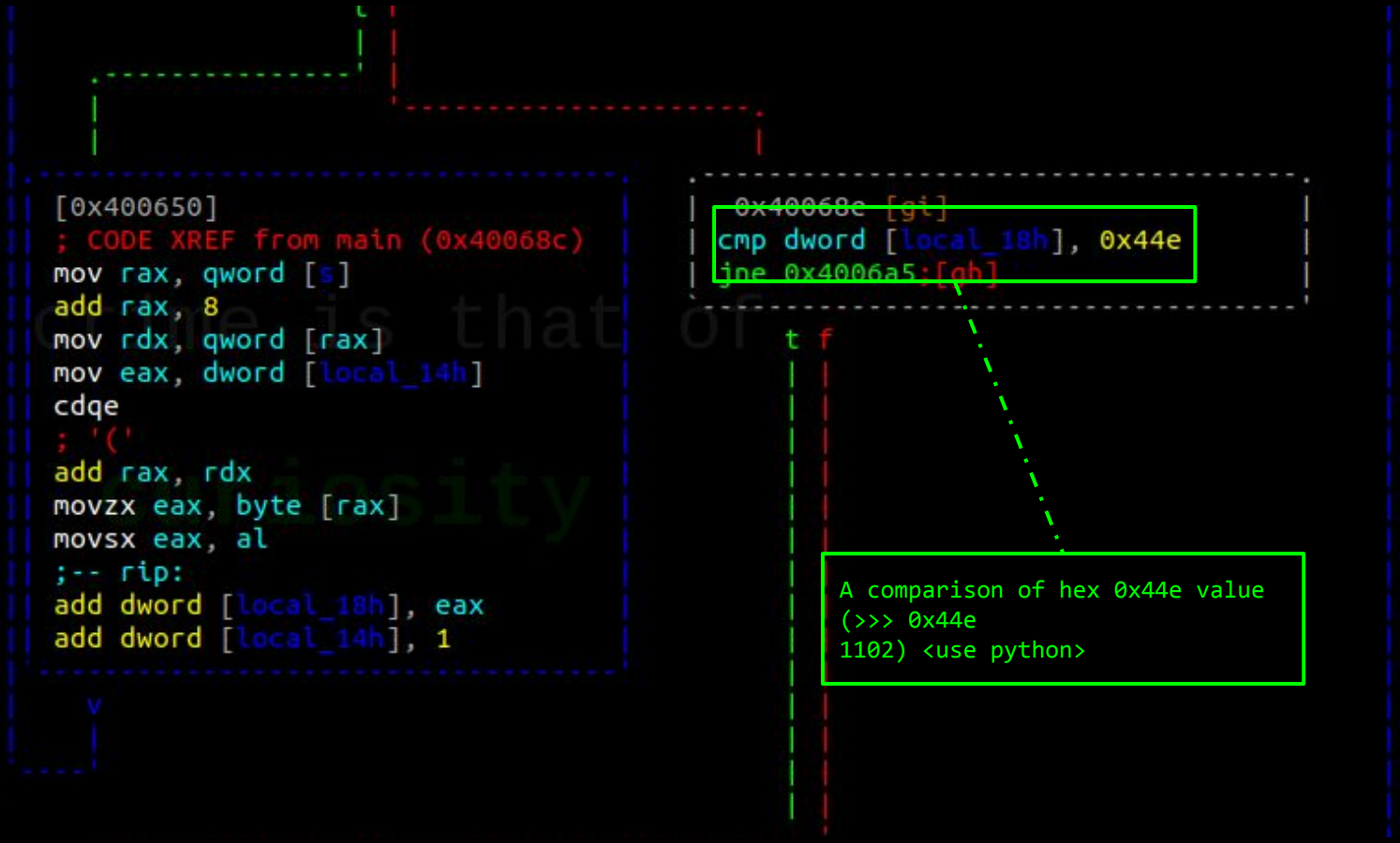


{ reverse engineering }



{ reverse engineering }

{ reverse engineering }



{ reverse engineering(contd.) }

Observations
>> two variables named-- local_14h and local_18h are set to zero at two blocks
>> there exists a loop and local_14h is increased by 1 at the end of the loop
<rename them by going to command mode with : afvn <previous name> <new name> if you want for better understanding>
>> A strlen compare is called and upon false condition stops the loop
>> There exists two similar assembler instruction in the looped blocks
>> A local_18h address's value is compared with 1102 (0x44e)
Let's inspect block b

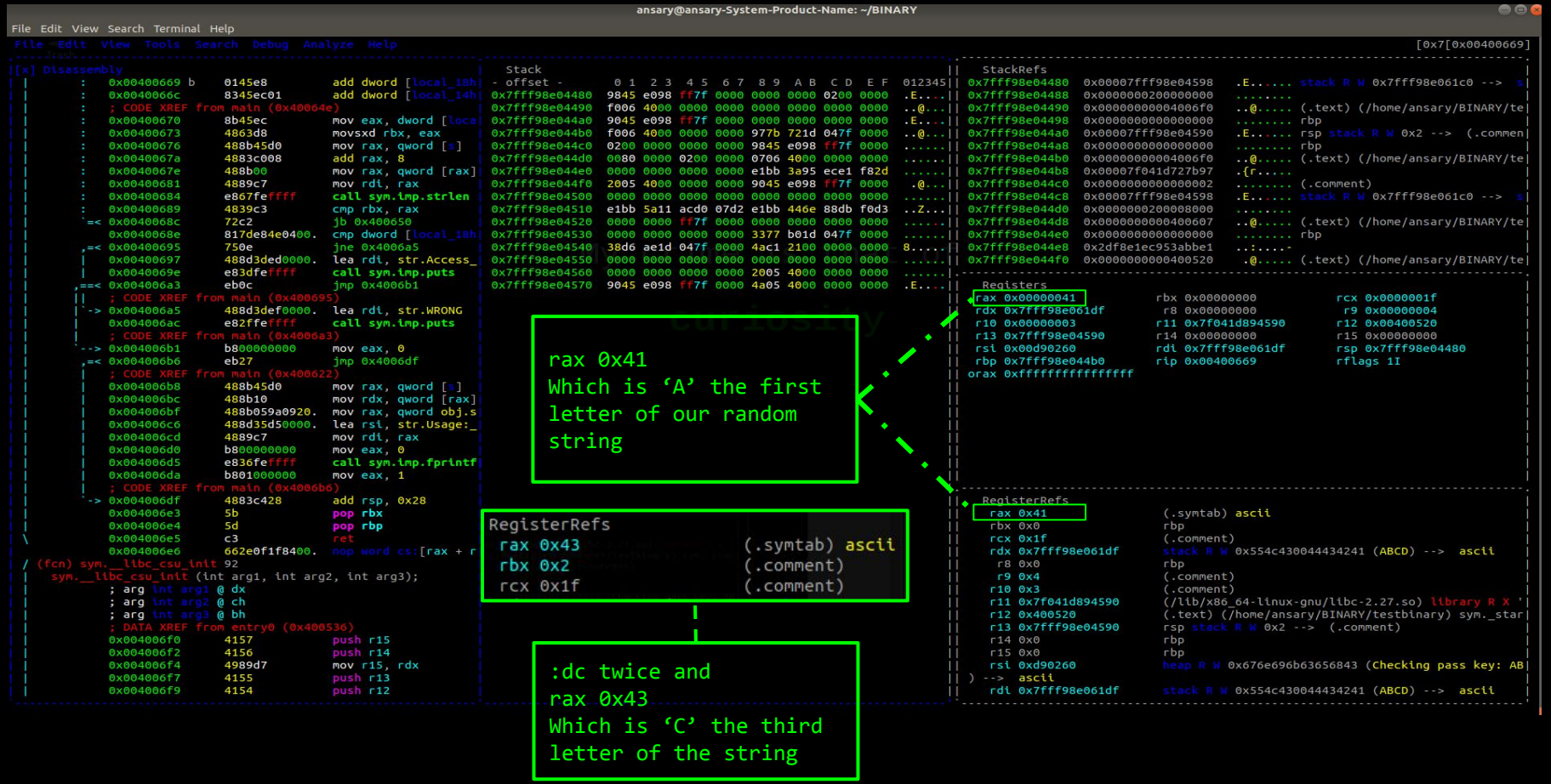
```
mov rax, qword [s]
add rax, 8
mov rdx, qword [rax]
mov eax, dword [local_14h]
cdqe
; '('
add rax, rdx
movzx eax, byte [rax]
movsx eax, al
add dword [local_18h], eax
add dword [local_14h], 1
```

>> add rax, 8 -- 8 is added in both cases >> meaning an address was loaded
<because we have 64 bits and we often divide memory in 8 byte chunks
Imagine an array in memory, which uses multiple chunks.
The first address simply points to first value in the array.
To get the second value we need to add 8 to the address >
>> mov rdx, qword [rax] -- after addition the value from an
address is loaded in rdx
<[] closed registers point to address and the value in that address is loaded>
>>in block b
>> mov eax, dword [local_14h] -- the counter variable local_14h is loaded into eax
>> add rax, rdx -- if rax pointed to the starting address of a string adding rdx will point to the next character
>> add dword [local_18h],eax -- adding rax(eax) to local_18h which was set to 0
In conclusion -- There exist a loop that adds the characters of a string to a variable until the string length is reached. Then the total length is compared to a specific value

Now let's practically visualize if this this assumption is true or not

>> set the breakpoint in the address where add dword [local_18h],eax is executed
(press p address view ,[address]> ood ABCD, [address]> db <breakpoint)
>> V! Observe rax

{ reverse engineering(contd.) }



{ keygen }

```python
#!/usr/bin/env python3
import random
import string
def check_key(key):
    char_sum=0
    for char in key:
        char_sum+=ord(char)
    return char_sum
def gen_valid_keys(key_value=1102,size=20, chars=string.ascii_letters + string.digits,num=10):
    i=0
    key=''
    while True:
        key +=random.choice(chars)
        char_sum=check_key(key)
        if char_sum>key_value:
            key=''
        elif char_sum==key_value:
            print("found valid key:{}".format(key))
            i+=1
        if i==num:
            Break
if __name__=='__main__':
    gen_valid_keys()
```

```
# Remember the number 1102 (0x44e)??
>>key_gen <write your own>
```

```
$ python3 keygen.py
found valid key:hODgCDyMPMByG
found valid key:OG9ajO3FQQisn
found valid key:Rm43JIqh4GWcS4
found valid key:DvAgoQB2uW2vD
found valid key:qyEdGPqzGRZF
found valid key:uUAY2TRxtxx6
found valid key:dowG908Ks5Fjy
found valid key:Iw3FFitqFnyT
found valid key:sN0SfHdCX8mdT
found valid key:hmo5SKnOp5li
```

{ parser differential }

```
>> executed by linux but gdb or radare fails to execute
# remember the kernel parser execve ?
>> one way is fuzzing
# in c source code-
>> change fprintf(stderr,"Usage: %s  <key>\n",argv[0]); to printf("Usage: %<name>  <key>\n");
>> change return 1; to return 0;
# {the return values indicate exit code and you can check the exit status by ‘echo $?’ in terminal  }
>> recompile as a ELF 64-bit LSB shared object (not executable i.e-- remove -no-pie) if needed

>> pipe the proper output and check it with any valid key by the keygen
$ ( ./testbinary hODgCDyMPMByG ; ./testbinary) > orig_output
$ cat orig_output
>> pipe the disassemble main command into gdb and check if gdb is able to disassemble the original file
$ echo disassemble main | gdb testbinary > orig_gdb
$ cat orig_gdb
>> pipe analyze all,seek main and print disassembly command into radare and check radare output
$ echo -e "aaa\ns sym.main\npdf" | r2 testbinary > orig_radare
$ cat orig_radare

# now our goal is to create a fuzzed file that gives the orig_output
# but fails to produce the orig_gdb and orig_radare
```

{ parser differential (contd.) }

```python
#!/usr/bin/env python3
import os
import random
def flip_byte(in_bytes):
   i=random.randint(0,len(in_bytes))
   c=bytes([random.randint(0,0xFF)])
   return in_bytes[:i]+c+in_bytes[i+1:]

def copy_executeable(file_name='testbinary'):
   os.system('cp {} {}_fuzzed'.format(file_name,file_name))
   with open (file_name,'rb') as orig_file , open(f'{file_name}_fuzzed','wb') as fuzzed_file:
       fuzzed_file.write(flip_byte(orig_file.read()))


def compare_output(o1,o2):
   with open (o1) as f1,open (o2) as f2:
       if (f1.read()==f2.read()):
           flag=True
       else:
           flag=False
       return flag
```

{ parser differential (contd.) }

```
def check_fuzzed_output(file_name='testbinary',orig_output='orig_output',key='hODgCDyMPMByG'):
     os.system( f'(./{file_name}_fuzzed {key}; ./{file_name}_fuzzed) > fuzzed_output')
   return compare_output(orig_output,'fuzzed_output')
def check_fuzzed_radare(file_name='testbinary',orig_radare='orig_radare'):
   os.system( f'echo -e "aaa\ns sym.main\npdf" | r2 {file_name}_fuzzed > fuzzed_radare')
   return compare_output(orig_radare,'fuzzed_radare')
if __name__=='__main__':
   while True:
       copy_executeable()
       output_flag=check_fuzzed_output()
       if output_flag:
           print('fuzzed binary linux executable')
           radare_flag=check_fuzzed_radare()
           if not radare_flag:
               os.system('tail fuzzed_radare')
               satisfied=input('Satisfied?(y/n)')
               if satisfied=='y':
                   break
   print('nicely done :)')
```

{ parser differential (contd.) }

>> now manually check if our new fuzzed binary is indeed executed by linux but can not be opened by radare

SINCE THAT WAS A SUCCESS

Your task is to the same for gdb and radare both

Hint:

```
def check_fuzzed_gdb(file_name='testbinary',orig_gdb='orig_gdb'):
   os.system( f'echo disassemble main | gdb {file_name}_fuzzed > fuzzed_gdb')
   return compare_output(orig_gdb,'fuzzed_gdb')
```

Add a checking for both gdb and radare

>> you could face segmentation faults -- google how to avoid them

{long break }

>> Now we will dive into the linux kernel and see what it means to have an operating system

Heads up

{It’s going to be tougher than what we have seen till this point <combined> }

>> So fresh up a bit and we will continue

{syscalls}

```
$ man syscalls
DESCRIPTION
       The system call is the fundamental interface between an application and the Linux kernel.

>> remember strace ??-- that traced system calls and showed a function called write instead of printf??
$ man 2 write
DESCRIPTION
       write() writes up to count bytes from the buffer starting at buf to the file referred to by the file
descriptor fd.
>> void main () {
write(1,"HACK\n",5);}
>> compile
>> open with radare in debug mode (aaa,s sym.main,pdf)
>> set breakpoint(db) at call.sym.imp.write
>> enter visual mode (V!)
>> start the program(:dc)
>> press s to step through the code with rip
# see plt?? (procedure linkage table/ function trampoline ) -- just remember the name(search google to learn more)
>> step through the code from libc library
>> it will take some time.....to reach syscalls (next slide)
{note: this is why we used shift+s before because using only s will step into syscalls}
```

{syscalls(contd.)}

{syscalls(contd.)}

>>From intel assembler reference: this is a fast call to privilege level 0 system procedures and the OP code is 0F 05
>>It loads rip from IA32LSTR_MSR (MSR=MODEL SPECIFIC REGISTER)
<It's like jump that loads rip from another point in this case MSR>
>> The address was loaded during boot through WRMSR instruction and you have to have level 0 privilege to access this
>> so you can't access this now by c program which is prev. Level-3
(technically you can access with syscalls but then you can't control what will be executed at level 0 because it will jump to a predefined address)
>> a value was loaded in eax : (0x15 in this case)

```
0x7f5c2cb85c74      b815000000      mov eax, 0x15
0x7f5c2cb85c79      0f05            syscall
```

>> Then syscall happened to access level-0 by jumping into a fixed address in kernel
[if you think we got this you are wrong by a mile]
{because there are other syscalls for the kernel taking the number from the register}
{and executing write function the assembly code will say mov eax, 0x1 then syscall with op code 0F 05, so you can try and find it but it's not necessary to understand the concept}
>> NOW!! the kernel knows it has to execute write function that is in read_write.c
https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/fs/read_write.c
>> just search 'SYSCALL_DEFINE3(write' in the code and see what happens when the write is called
(by the way the read_write.c was written by Linus Torvalds the father of linux himself)

{syscalls(contd.)}

https://static.lwn.net/images/pdf/LDD3/ch03.pdf

>> go to read and write section (LDD - 'Linux Device Drivers' tells in details how kernel works)

"The code for read and write in scull needs to copy a whole segment of data to or from the user address space. This capability is offered by the following kernel functions,which copy an arbitrary array of bytes and sit at the heart of most read and write implementations:

```
unsigned long copy_to_user(void __user *to,const void *from,unsigned long count);
unsigned long copy_from_user(void *to,const void __user *from,unsigned long count);
```

"-- from the book

>> so what does the user address space mean?

https://elixir.bootlin.com/linux/latest/ident/copy_from_user

>> go to linux cross reference and search copy_from_user identifier to look how it is defined for various architectures and what it looks like to see an OS function

Task (20 min)

>> Explore the flow of function calls till we reach get_user_asm which is a preprocessor macro

{HINT: click on the function call and see where they lead also don't forget to choose correct arch. }

>> don't be afraid -- before compiler turns this into machine code its standard C

>> the assembly syntax you will see in code is at&t

>> Well that was easy -skip 'this and that' it just executes a mov instruction… right???

>> NO -- the magic is happening somewhere else

>> when the kernel tries to execute the instruction it will cause page fault

BECAUSE IT TRIES TO ACCESS A VIRTUAL ADDRESS

>> remember the ASAM_STAC and ASAM_CLAC in get_user_asm code? Well just below that there is a line of code defined how the kernel handles hardware exceptions (with MMU)

{concluding remarks(Q/A)}

>> The base purpose is to develop a very basic idea on the topics
>> Also breaking the stereotype that hacking is all about fancy stuff .
>> IT’s simply about knowing I.T. on a deeper level
>> solve CTF’s (capture the flag)
>> learn about stack buffer overflows,memory corruption, shell code,heap overflows, format string exploits e.t.c

<<<<<THERE IS STILL A LOT>>>>>>>>
HAPPY LEARNING

{<(thank you)>}