CSIT 5740 Introduction to Software Security

Note set 4B

Dr. Alex LAM

THE HONG KONG UNIVERSITY OF SCIENCE AND TECHNOLOGY

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

The set of note is adopted and converted from a software security course at the Purdue University by Prof. Antonio Bianchi

Address Space Layout Randomization (ASLR)

Guessing Addresses

- Overflow a buffer → Overwrite a pointer (e.g., return address on the stack) → Make the execution jump to a memory region which:
  1) we control the content of
  2) is executable (more on this later)
  3) **we know where it is located**

Guessing Addresses

- In most cases the address of the buffer is not known
- It has to be “guessed” (and the guess must be VERY precise)
- The stack address of a program can be obtained by using gdb
    - For now we assume: no memory layout randomization

- Given the same environment and knowing the size of command-line parameters, the address of the stack can be roughly guessed

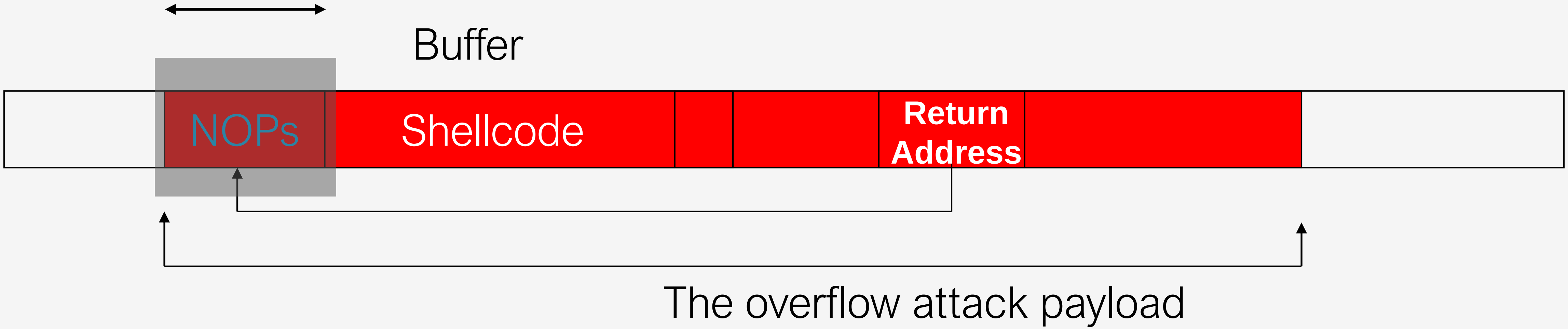
- However the exact value may be slightly different when a program is started by gdb or not (due to different environment variables)

- That’s why we use NOP sleds for writing the shellcode

NOP Sled

- Just like what have mentioned at the beginning of note set 4A. To increase the probability of success, we can add, at the beginning of our controlled buffer (containing the shellcode) a series of NOP instructions
  - In Intel assembly, NOP is 0x90

If we guess a value within this rage the shellcode will be executed successfully



Address Space Layout Randomization (ASLR)

- An Operating System mitigation technique → makes exploitation harder
- Randomizes the position of the:
  - Heap
  - Stack
  - Dynamically-linked libraries
  - Program’s main code, if compiled as position independent (PIE). Enabled/Disabled with compiler options -fPIE/-no-pie
    slightly slower

Address Space Layout Randomization (ASLR)

- On 32-bit Linux: only 20 bits of entropy
  - Still vulnerable to brute-force attack, if unlimited attempts are possible
  - NOP-sled can help the attack significantly

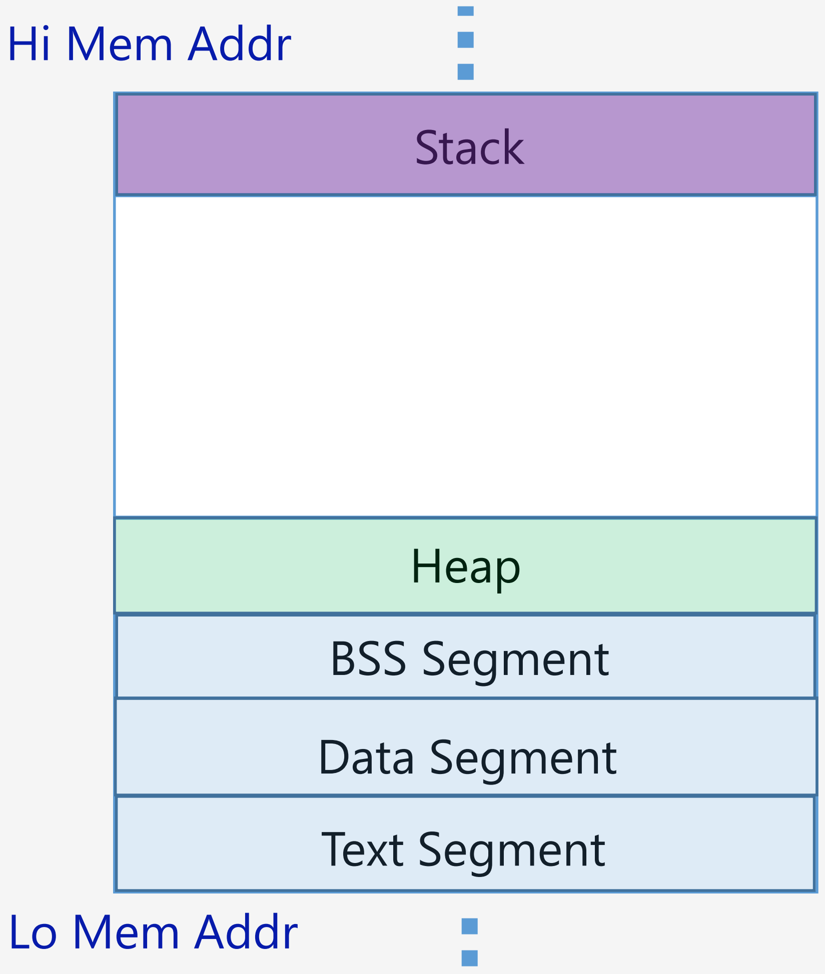
- 64-bit architectures: more virtual addresses available
  → more randomization → much more secure

ASLR in Linux

- ASLR can be disabled by:
  - Setting the associated kernel variable to 0
  - echo "0" > /proc/sys/kernel/randomize_va_space
    - Requires root
- ASLR can be enabled by (it is enabled by default)
  - echo “2" > /proc/sys/kernel/randomize_va_space
    - Also requires root
- Running a program in GDB
  - makes the program ignoring the setuid bit
  - disables randomization by default

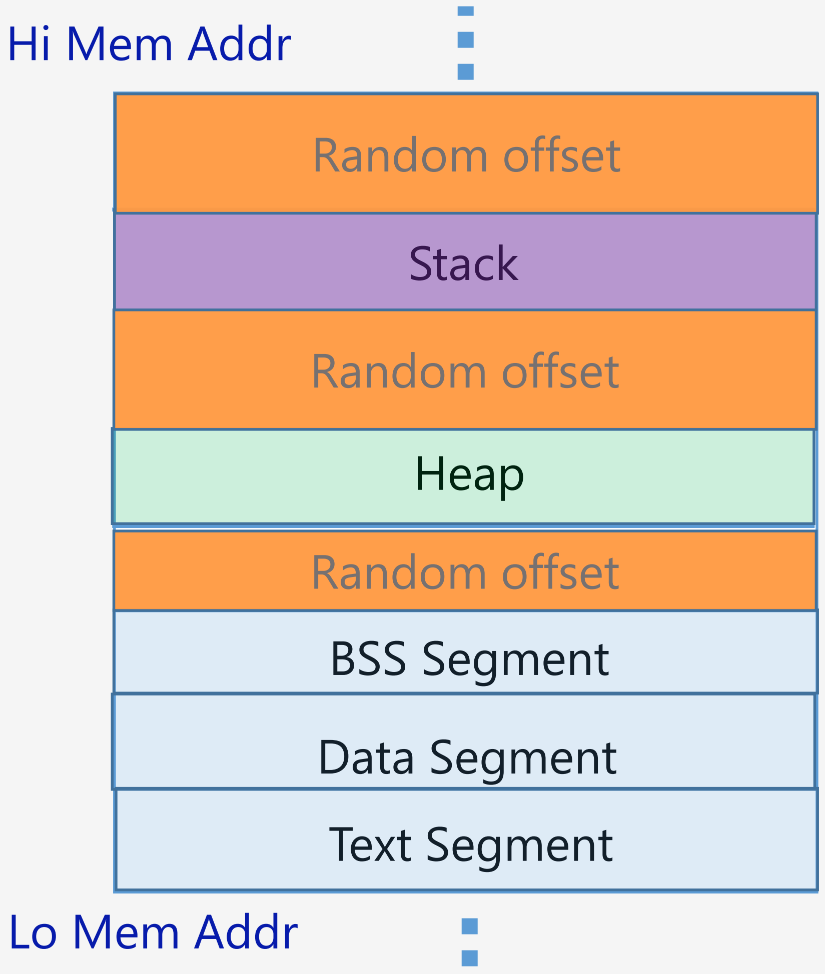
Example: ASLR disabled

When there is no ASLR, x86 memory looks like this



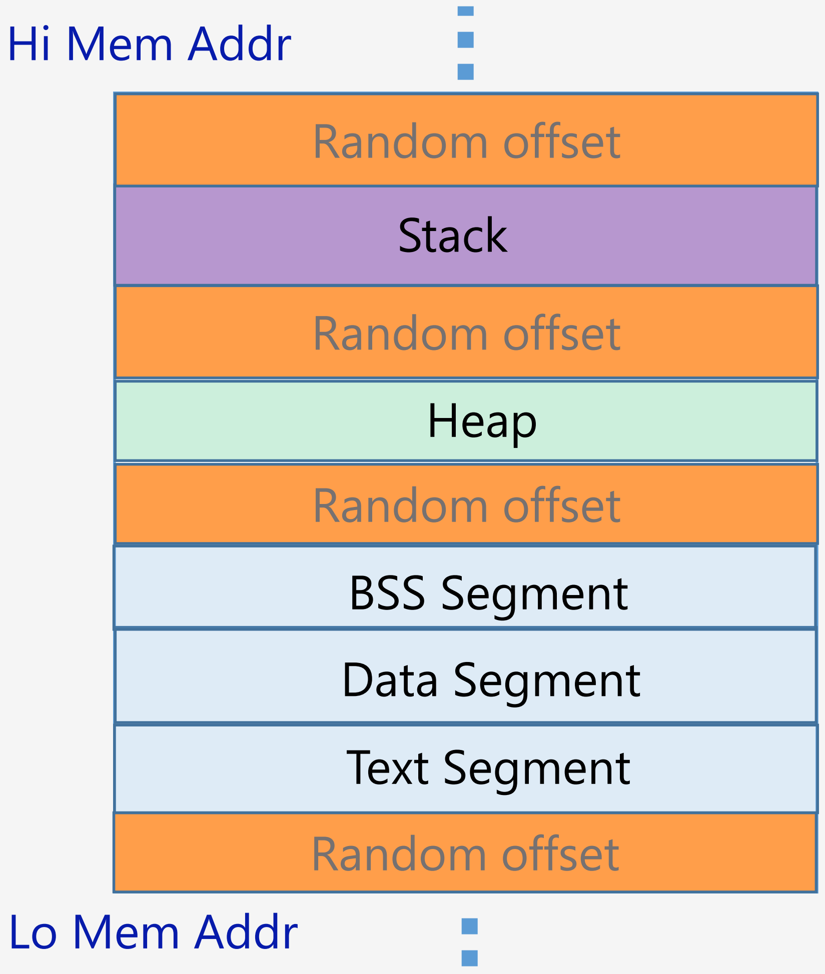
Example: ASLR enabled, non-PIE binary

When there is there ASLR, but PIE is turned off



Example: ASLR enabled, PIE binary

When there is there ASLR, PIE is also turned on



ASLR Exploitation

- If the randomization is limited (32 bit)
  - Bruteforcing
  - NOP Sled
  - …

- fork does not re-randomize the layout → it may help bruteforcing
  - e.g., in the case of a “forking server”
  - fork does not change the memory locations used by the child process → after fork, the memory layout of the child will be equal to the memory layout of the parent process

ASLR Exploitation

- If bruteforcing is not feasible, we need “a leak”
  - For instance, we can force the program to print the value of a pointer → we can use this value to know where the code has been loaded at
  - Alternative, we can exploit a format string vulnerability (more on this later)

- If we need a leak, an exploit becomes more complex.
  At least 2 steps are required, example:
  1) Read memory → Leak a pointer → Compute where our controlled buffer is
     - Note that the “relative distance” between many pointers normally remain constant even in the presence of ASLR
       → Leaking one pointer allow to recover the address of other pointers
  2) Write memory → Overwrite a function pointer → Jump to the controlled buffer

Format String attack to leak information

Leaking Memory with printf

- `int printf(const char *format, ...);`

- Notice that “. . .” above? It means the printf() function takes in an variable number of arguments (i.e. variadic).
  - How does it know how many arguments are passed to it?
  - It infers from the first argument *format, which is a string
  - For example: `printf("Hello %s, you are %d years old\n", name, age);`

- What happens when we supply just one argument instead of two?
  - `printf("Hello %s, you are %d years old\n", name);`
  - `printf()` will output the string (`%s`) according to the `rdi` value (the computer has copied the data in the name variable into it), and then it will output the `rsi` value (for the unsupplied `%d`)

printf Internals

- To find out why, we must first understand what happens when we call
- printf("%d", 1); // How does this work?
- printf assumes that a parameter was passed, so it accesses the next location that should have one: in this case, rsi (second parameter)
- Keep in mind the calling convention:
  - The arguments of the function calls are copied to : rdi, rsi, rdx, rcx, r8, r9, < stack. . . >

printf("%d", 1)

rdi:	addr of the format string "%d"
rsi:	1
rdx:	garbage
rcx:	garbage
r8:	garbage
r9:	garbage

printf("%d%d", 1, 2)

rdi:	addr of the format string "%d%d"
rsi:	1
rdx:	2
rcx:	garbage
r8:	garbage
r9:	garbage

```
printf("%d%d%d", 1, 2, 3)
```

rdi:	addr of the format string "%d%d%d"
rsi:	1
rdx:	2
rcx:	3
r8:	garbage
r9:	garbage

```
printf("%d%d%d%d", 1, 2, 3, 4)
```

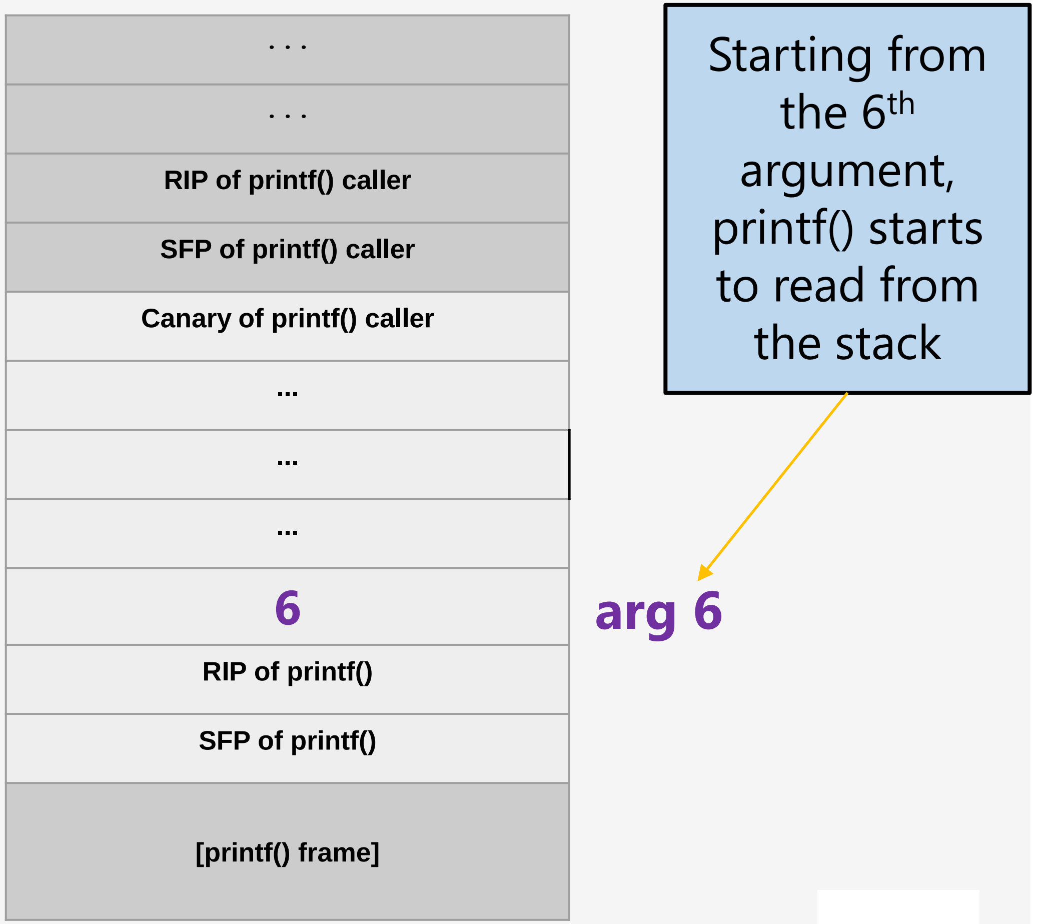
rdi:	addr of the format string "%d%d%d%d"
rsi:	1
rdx:	2
rcx:	3
r8:	4
r9:	garbage

```
printf("%d%d%d%d%d", 1, 2, 3, 4, 5)
```

rdi:	addr of the format string "%d%d%d%d%d"
rsi:	1
rdx:	2
rcx:	3
r8:	4
r9:	5

printf("%d%d%d%d%d%d", 1, 2, 3, 4, 5, 6)

rdi:	addr of the format string "%d%d%d%d%d%d"
rsi:	1
rdx:	2
rcx:	3
r8:	4
r9:	5



. . .

. . .

RIP of printf() caller

SFP of printf() caller

Canary of printf() caller

...

...

...

6

RIP of printf()

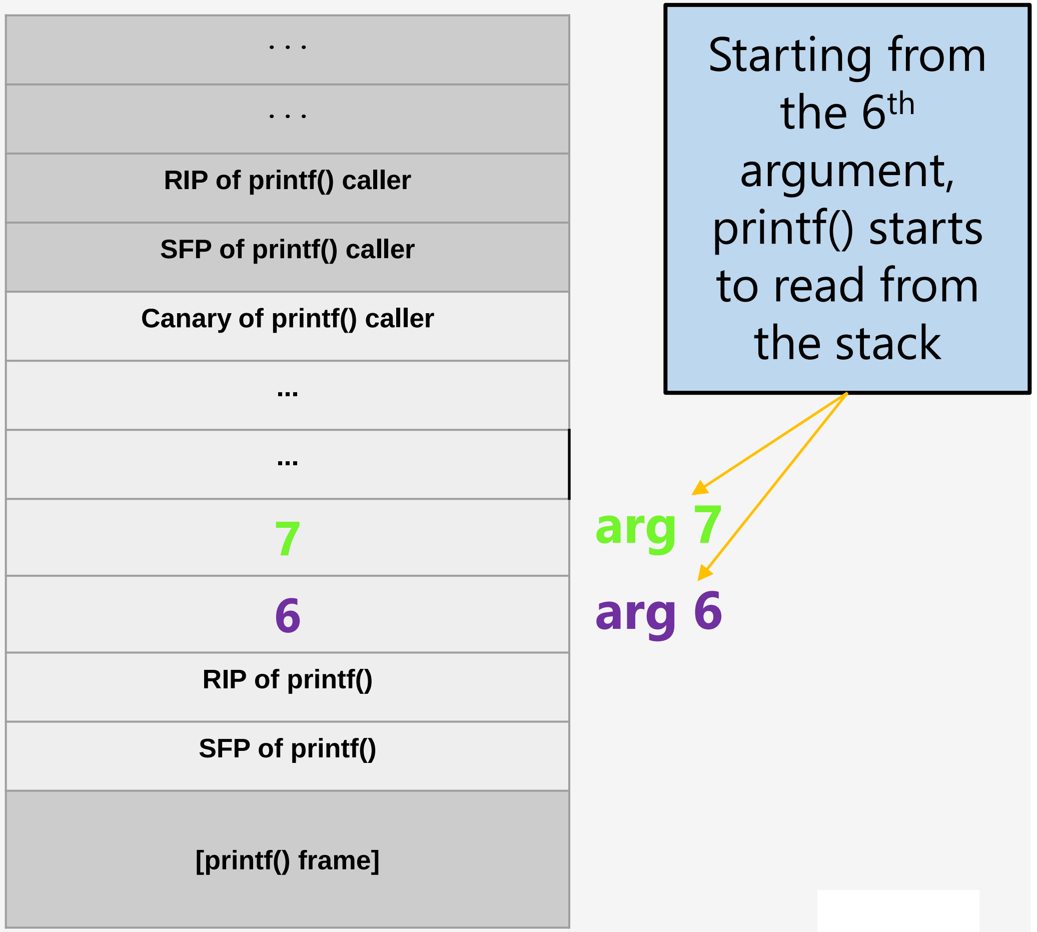
SFP of printf()

[printf() frame]

arg 6

Starting from the 6th argument, printf() starts to read from the stack printf("%d%d%d%d%d%d%d", 1, 2, 3, 4, 5, 6, 7)

rdi:	addr of the format string "%d%d%d%d...%d"
rsi:	1
rdx:	2
rcx:	3
r8:	4
r9:	5

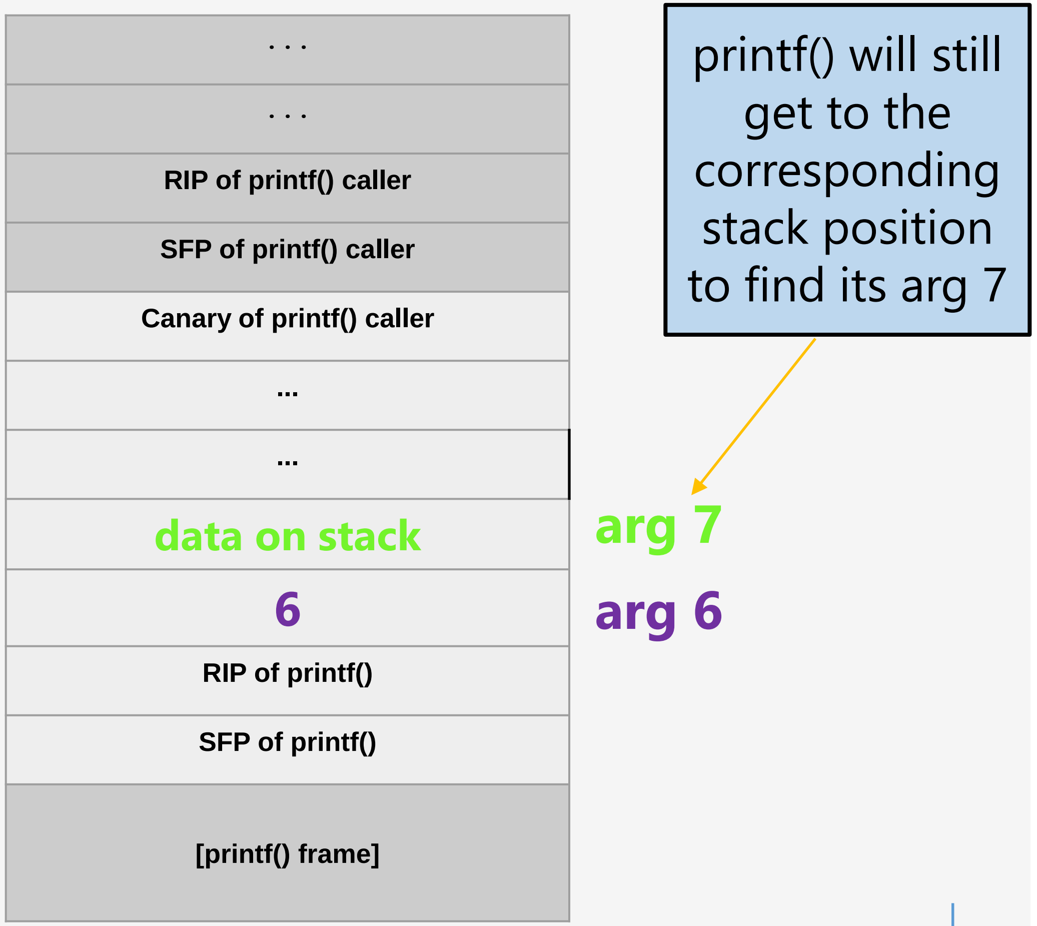


Starting from the 6th argument, printf() starts to read from the stack

What if we do not supply all the args

printf("%d%d%d%d%d%d%d", 1, 2, 3, 4, 5, 6)

rdi:	addr of the format string "%d%d%d%d...%d"
rsi:	1
rdx:	2
rcx:	3
r8:	4
r9:	5



. . .

. . .

RIP of printf() caller

SFP of printf() caller

Canary of printf() caller

...

...

data on stack — arg 7

6 — arg 6

RIP of printf()

SFP of printf()

[printf() frame]

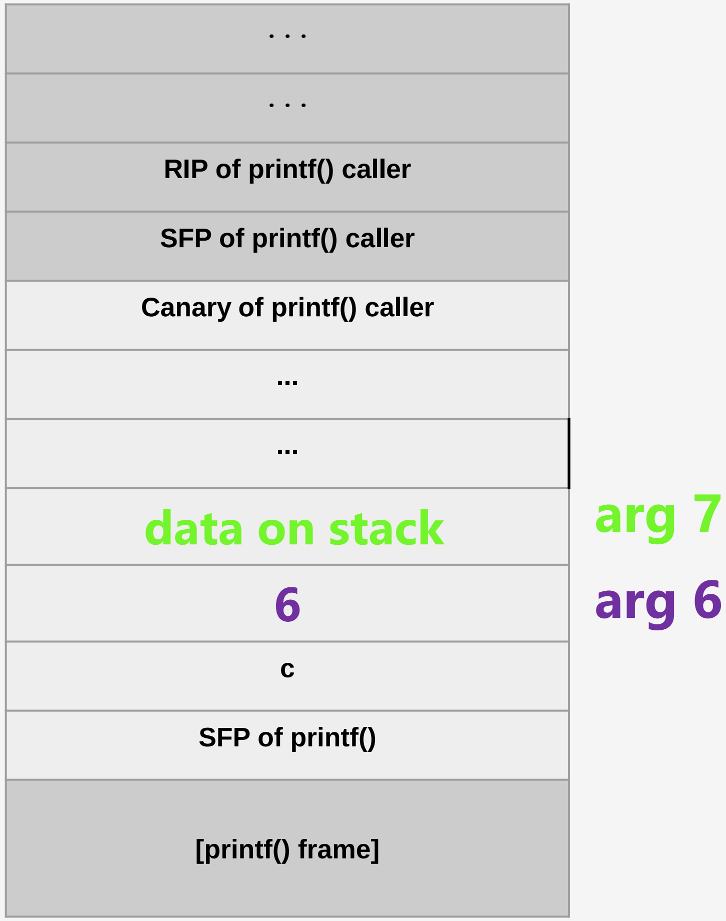
printf() will still get to the corresponding stack position to find its arg 7

What if we do not supply all the args

```
printf("%d%d%d%d%d%d%d", 1, 2, 3, 4, 5, 6)
```

So, the above printf will print:

123456data_on_stack



. . .

. . .

RIP of printf() caller

SFP of printf() caller

Canary of printf() caller

...

...

data on stack

arg 7

6

arg 6

c

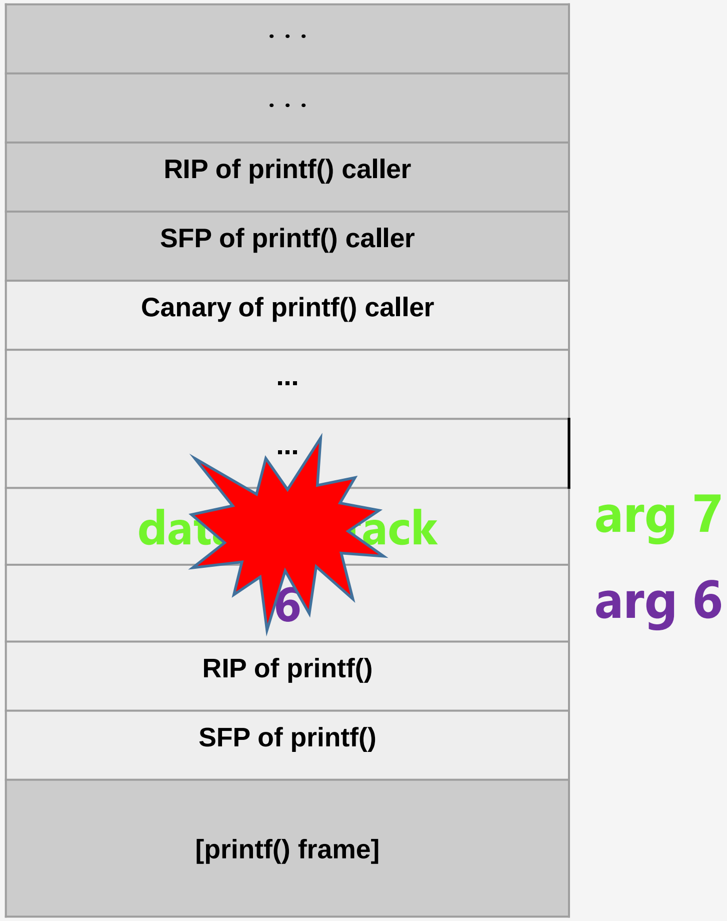
SFP of printf()

[printf() frame]

What if we do not supply all the args

printf("%d%d%d%d%d%d%d", 1, 2, 3, 4, 5, 6)

So, printf can be exploited to leak important data on the stack!



. . .

. . .

RIP of printf() caller

SFP of printf() caller

Canary of printf() caller

...

...

dat ack

6

RIP of printf()

SFP of printf()

[printf() frame]

printf: Exploitation

- If a program passes unsanitized user input to **printf**, we can cause it to utilize values from the stack that were not intended as parameters printf

- This sort of attacks is called: **format string attacks**

- Using this, we can dump stack information, e.g. via %p:

  1) Create buf as a global char array

  2) buf = "rsi: %p, rdx: %p, rcx: %p, r8: %p, r9: %p, stack: %p %p";

  3) printf(buf);

- This will print rsi, rdx, rcx, r8, r9, and two stack qwords
  → This allows us to leak data from the program

printf: Exploitation

- If a program passes unsanitized user input to **printf**, we can cause it to utilize values from the stack that were not intended as parameters printf

- It does not give direct control over the program, however
  - %p and other formats only let us **read** from the stack, not to write
    - It can be used to leak sensitive information from the program or defeat randomization

  - %n, can be used to **write** data!
    → disabled in many modern libc implementations

printf: Exploitation

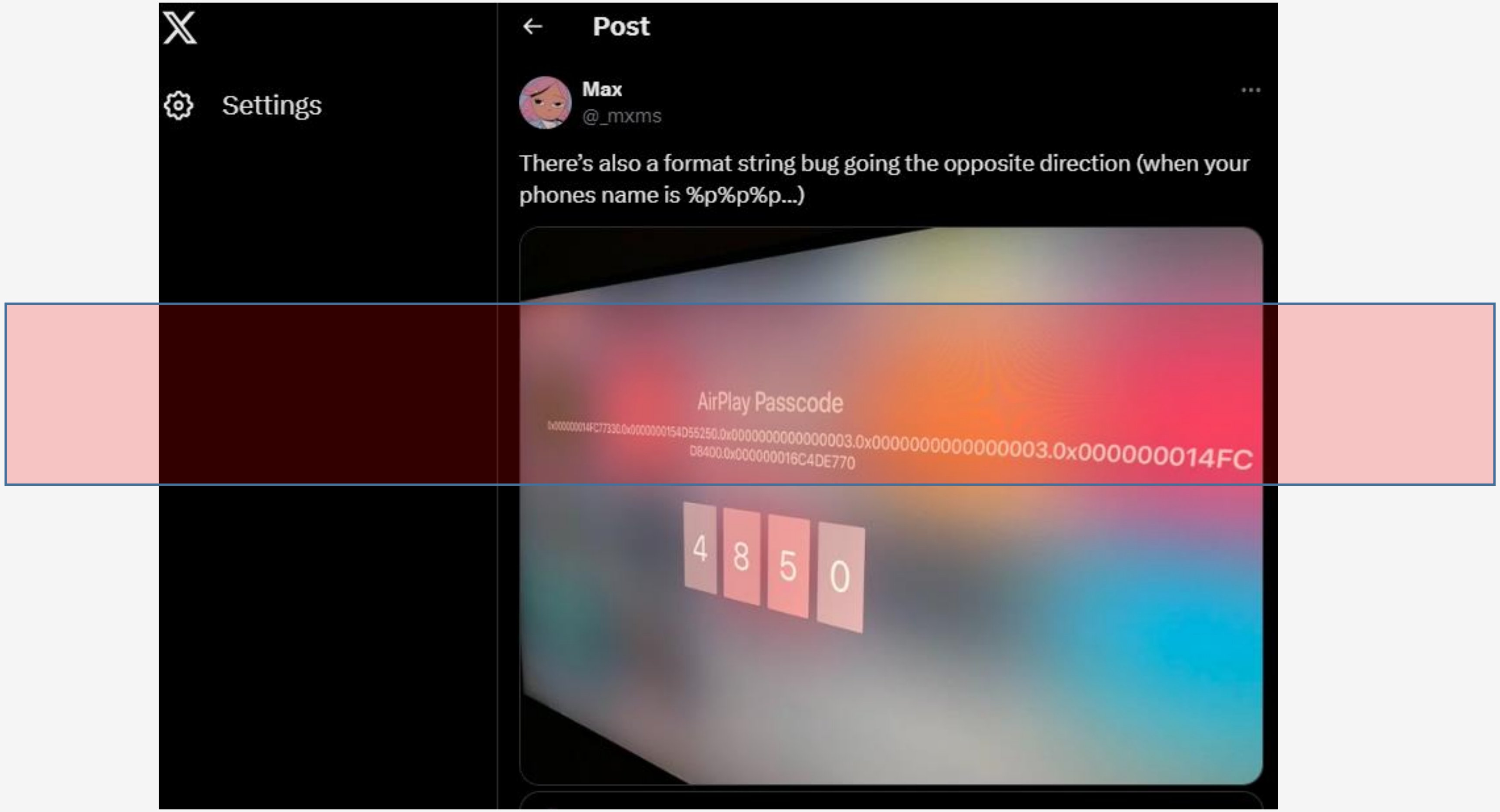
- **printf** supports the following format to print the nth argument in as a 32bit/64bit pointer on 32bit/64bit machines
  - %n$p
  - For example: printf(**"**%9$p**"**) will print the 9th argument
  - By using the above we will show how using the format string attack to defeat a stack memory protection mechanism called “stack canary”, which was supposed to protect the system against stack smashing

printf: Exploitation

- If we allow the user to supply arbitrary format string to **printf** , then the user can launch the format string attack. The following is one such scenario
  - Delcare a char array, say, buffer[]
  - Get the user input into buffer
  - Let printf print the user input directly (i.e. printf(buffer))
  - Then the user can provide strings like “%p%p%p” or “%9$p” to get the program information

- To mitigate that, one just need to use:
  - **`printf("%s", buffer)`**
  - The format string in the buffer will be considered static strings and will be output as the static strings “**%p%p%p**” or “**%9$p**”. It will not print any other data.

A real world example of format string attack on Apple Airplay in 2019



Max
@_mxms

There's also a format string bug going the opposite direction (when your phones name is %p%p%p...)

AirPlay Passcode

```
0x000000014FC77330.0x0000000154D55250.0x0000000000000003.0x0000000000000003.0x000000014FC
D8400.0x000000016C4DE770
```

4 8 5 0

Stack Canaries

Stack Canaries

- Add a **random** value before the saved returned address, and check it before the function returns

- If an overflow happens the canary's value change → the program's execution is halted before exploitation is possible

- Implemented by default in modern compilers by modifying the prologue/epilogue to add/verify the canary on the stack
    - Optimization: only used in functions using a buffer on the stack
    - Can be disabled with: -fno-stack-protector

Stack Canaries

- Miners protect themselves against toxic gas buildup in the mine with a canary
  - Canary: A beautiful bird that is sensitive to toxic gas
  - If toxic gas builds up, the canary dies first
  - The miners notice that the canary has died and evacuate the mine

Stack Canaries

- A canary value is unique every time the program **runs** but the same for all functions **within a run**

- A canary value uses a NULL byte as the first byte to mitigate string-based attacks (since it terminates any string before it (i.e. **the last byte, or the “end” is \x00**)
  - Example: A format string vulnerability with **`%s`** might try to print everything on the stack
    - The null byte in the canary will mitigate the damage by stopping the print earlier.

Stack Canaries

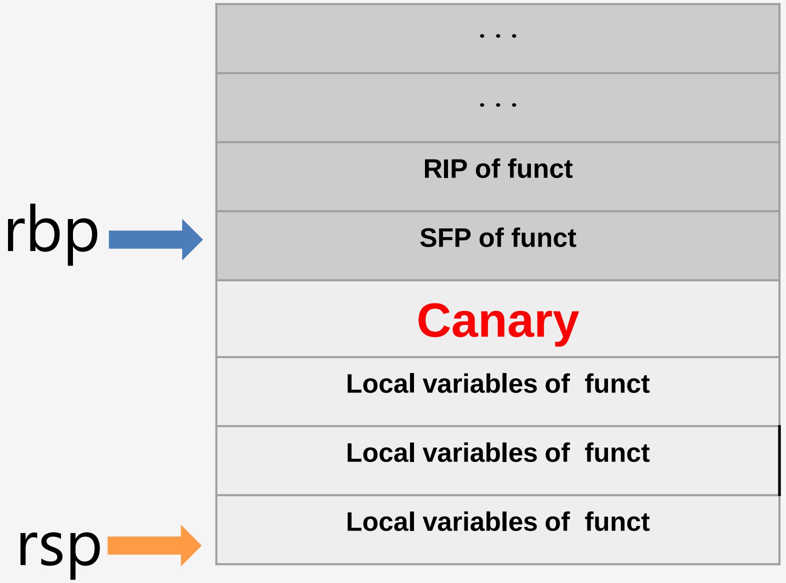
Prologue

```
push      rbp
mov       rbp,rsp
sub       rsp,0x20
mov       rax,QWORD PTR fs:0x28 ; fs:0x28 contains the canary
mov       QWORD PTR [rbp-0x8],rax
```

Epilogue

```
mov       rax,QWORD PTR [rbp-0x8]
xor       rax,QWORD PTR fs:0x28c ; if canary on stack same as the real
                                  ; canary, rax will be 0
je        400d4f; 0x400d4f is the address the leave instruction below
call       4007e0 <__stack_chk_fail@plt>; prints an error message and immediately terminates the program
leave
ret
```

Jump to here



. . .
. . .
RIP of funct
SFP of funct
Canary
Local variables of funct
Local variables of funct
Local variables of funct

Bypassing Canaries

- Memory Leaks (using format string attack)

- Arbitrary write
  - a[x] = y
  - just controlling x could be enough
  - x could be negative
  - x could be set as huge number and “wraps around”

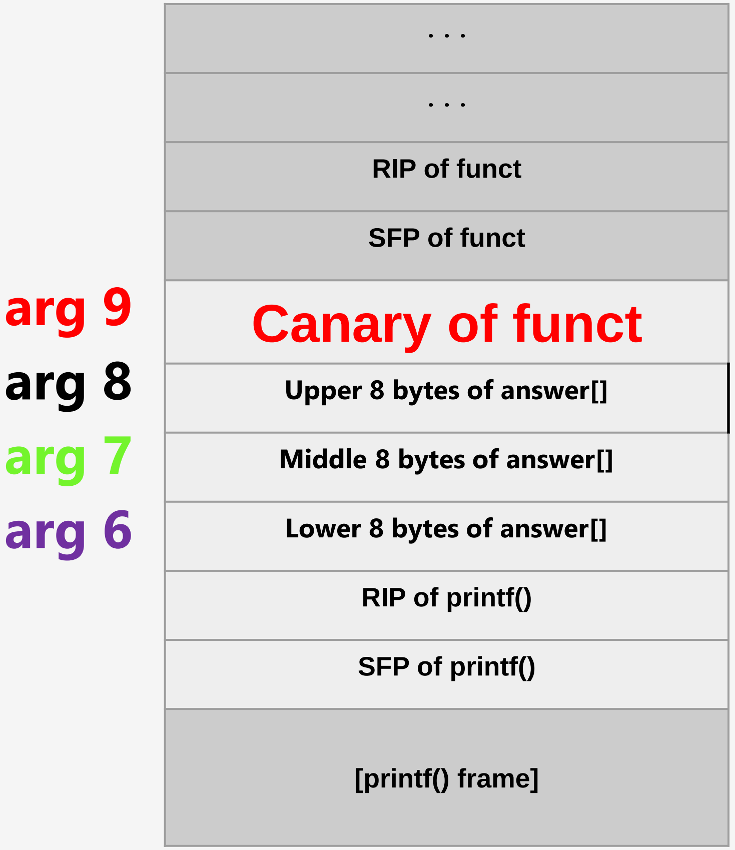
- Forking Servers and Bruteforcing

Exposing Canary through format string attacks

```
void funct() {
  char answer[24]=“%6$p,%7$p,%8$p,%9$p”;
  printf(answer)
}

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

  funct();
  return 0;
}
```



. . .

. . .

RIP of funct

SFP of funct

arg 9 — Canary of funct

arg 8 — Upper 8 bytes of answer[]

arg 7 — Middle 8 bytes of answer[]

arg 6 — Lower 8 bytes of answer[]

RIP of printf()

SFP of printf()

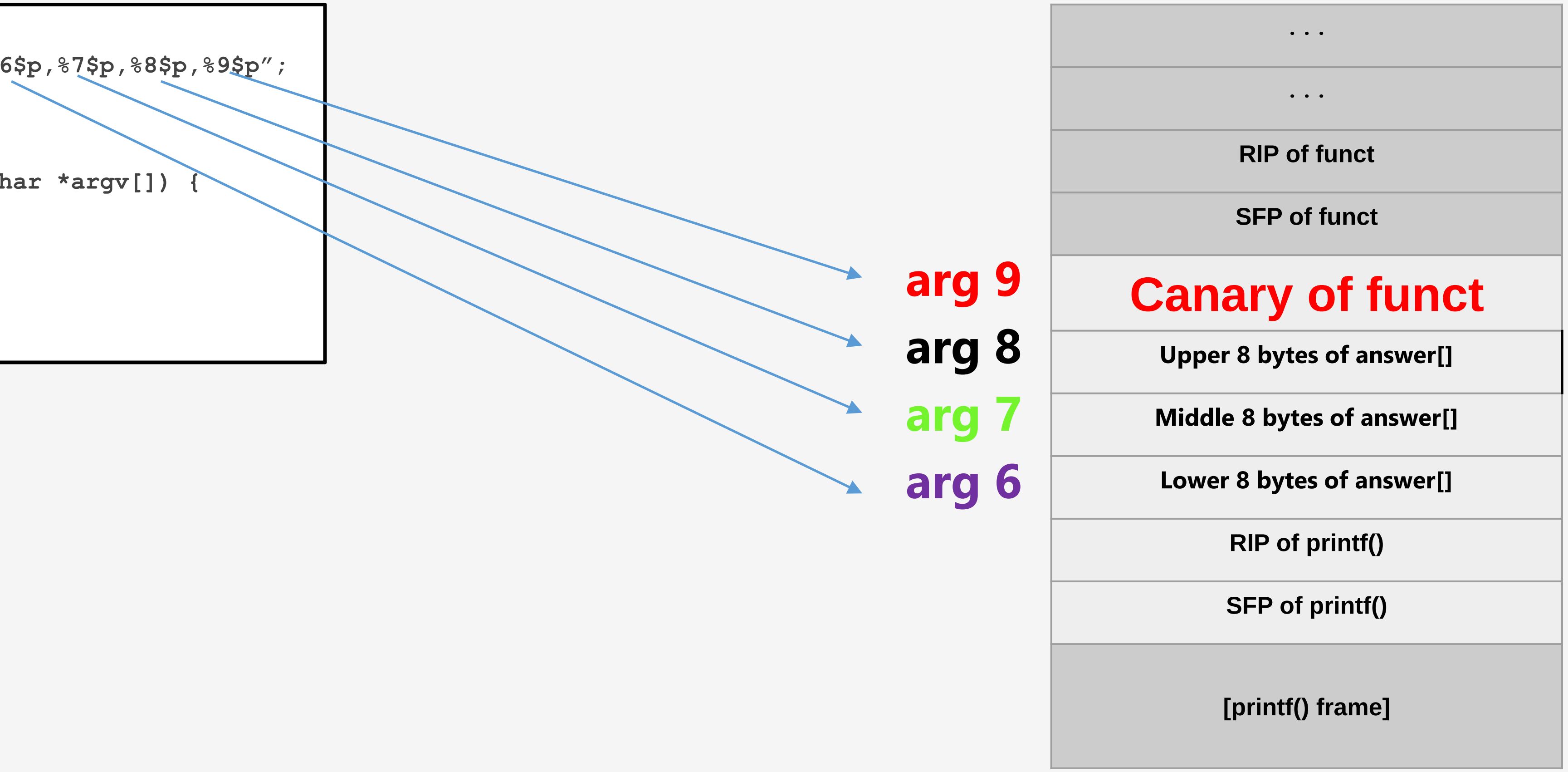
[printf() frame]

Exposing Canary through format string attacks

```
void funct() {
  char answer[24]=“%6$p,%7$p,%8$p,%9$p”;
  printf(answer)
}

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

  funct();
  return 0;
}
```



. . .

. . .

RIP of funct

SFP of funct

Canary of funct

Upper 8 bytes of answer[]

Middle 8 bytes of answer[]

Lower 8 bytes of answer[]

RIP of printf()

SFP of printf()

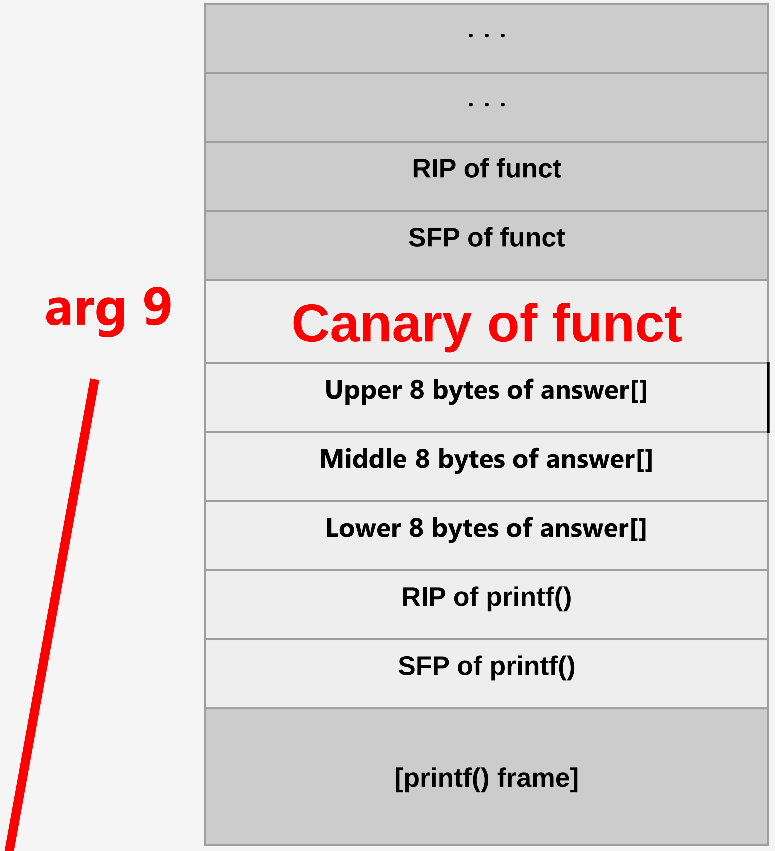
[printf() frame]

Exposing Canary through format string attacks

```
void funct() {
  char answer[24]=“%6$p,%7$p,%8$p,%9$p”;
  printf(answer)
}

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

  funct();
  return 0;
}
```



The output from the program could be:
0x2437252c70243625, 0x252c702438252c70, 0x702439, **0xd963a0cd2106600**

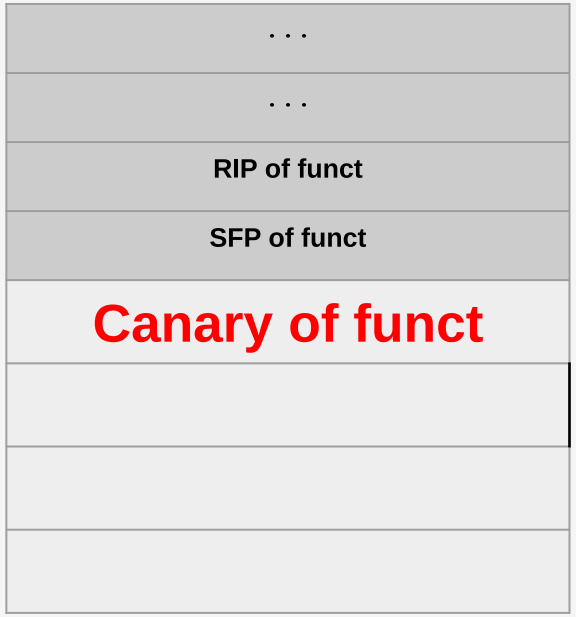
Canaries and Forking Servers

- As for ASLR, an interesting case is forking servers
  - fork does not change the layout of the memory (since it just “clones” the current process)

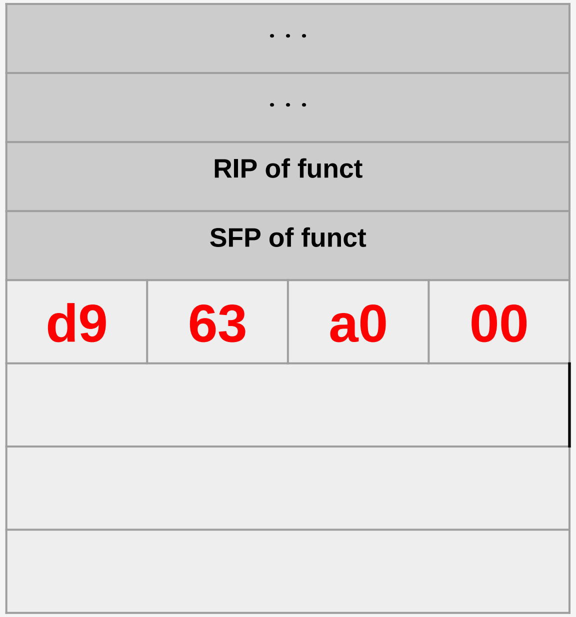
- the stack canary will not change when a program forks

- if we can overflow 1 byte at a time, we can try
  - to guess the canary byte-by-byte (max 256 tries per byte)
  - by checking if the child process crashed or not (e.g., the connection dropped)
  - Then in total, 256+256+256 guesses for 32-bit canary, and 256+256+…+256= 7 * 256 = 1792 guesses for 64-bit canary!

Guessing canary byte by byte (32-bit)



Guessing canary byte by byte (32-bit)



. . .

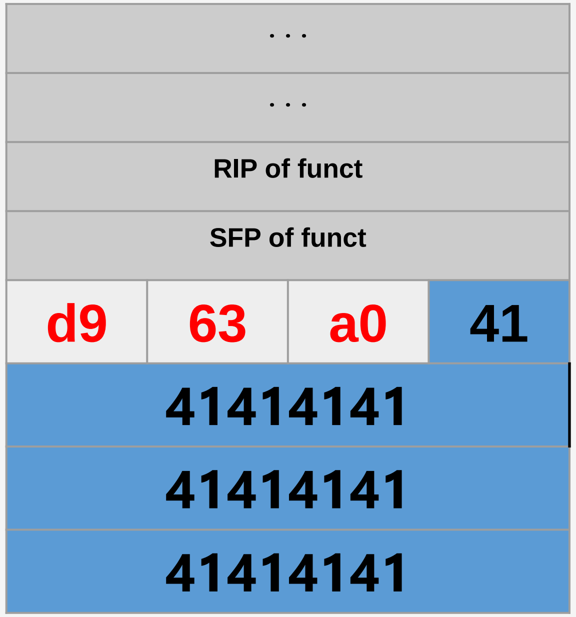
. . .

RIP of funct

SFP of funct

d9 63 a0 00

Guessing canary byte by byte (32-bit)



. . .

. . .

RIP of funct

SFP of funct

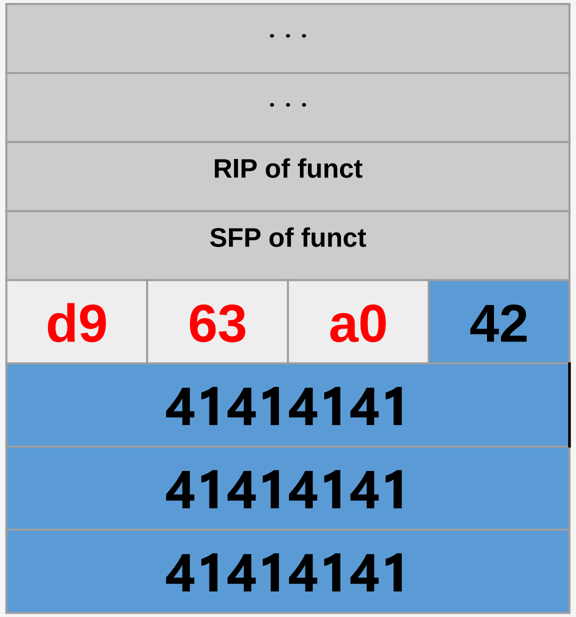
d9 63 a0 41

41414141

41414141

41414141

Guessing canary byte by byte (32-bit)



. . .

. . .

RIP of funct

SFP of funct

d9	63	a0	42

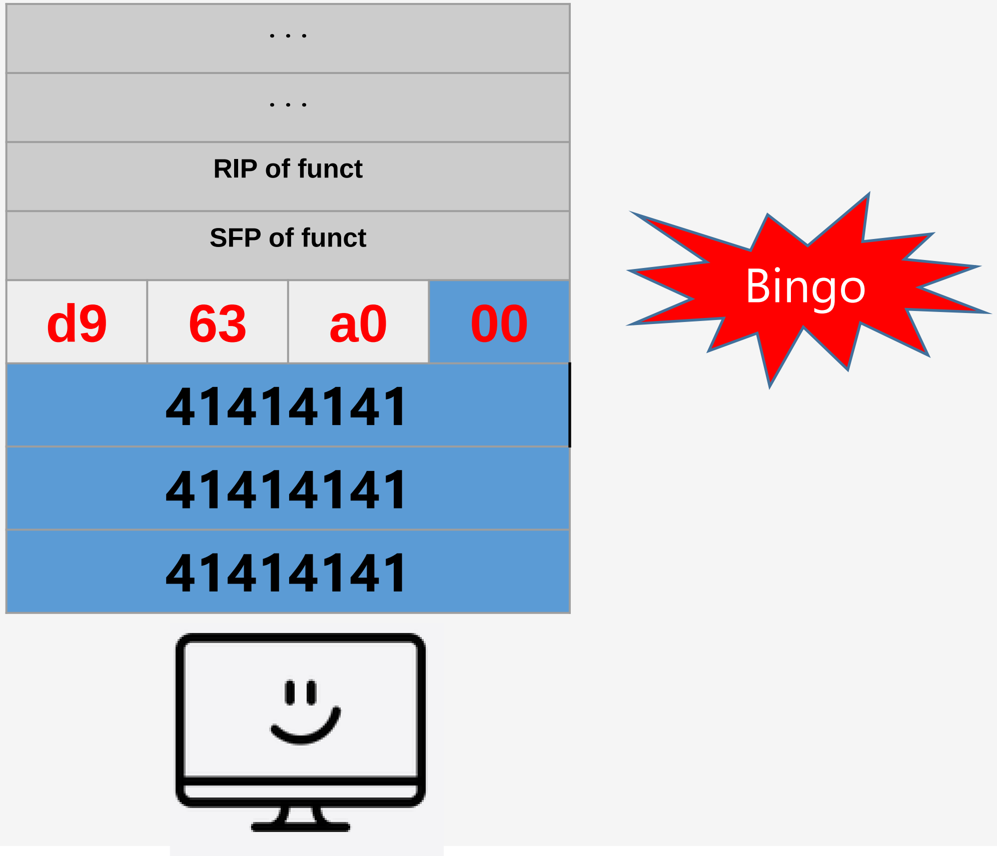
41414141

41414141

41414141

After at most 256 guesses…

Guessing canary byte by byte (32-bit)



. . .

. . .

RIP of funct

SFP of funct

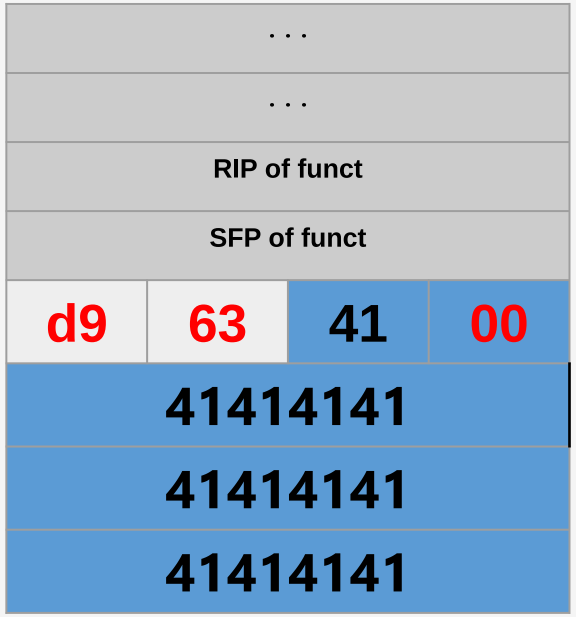
d9	63	a0	00

41414141

41414141

41414141

Guessing canary byte by byte (32-bit)



. . .

. . .

RIP of funct

SFP of funct

d9	63	41	00

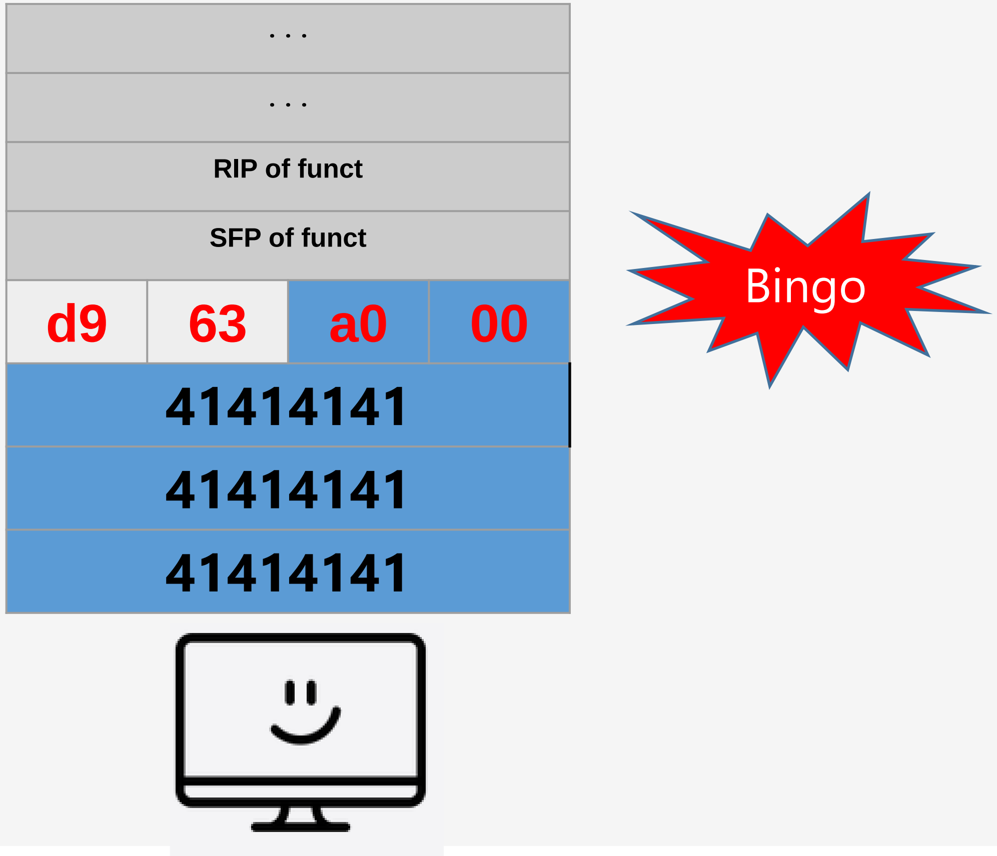
41414141

41414141

41414141

After at most 256 guesses…

Guessing canary byte by byte (32-bit)



. . .

. . .

RIP of funct

SFP of funct

d9	63	a0	00

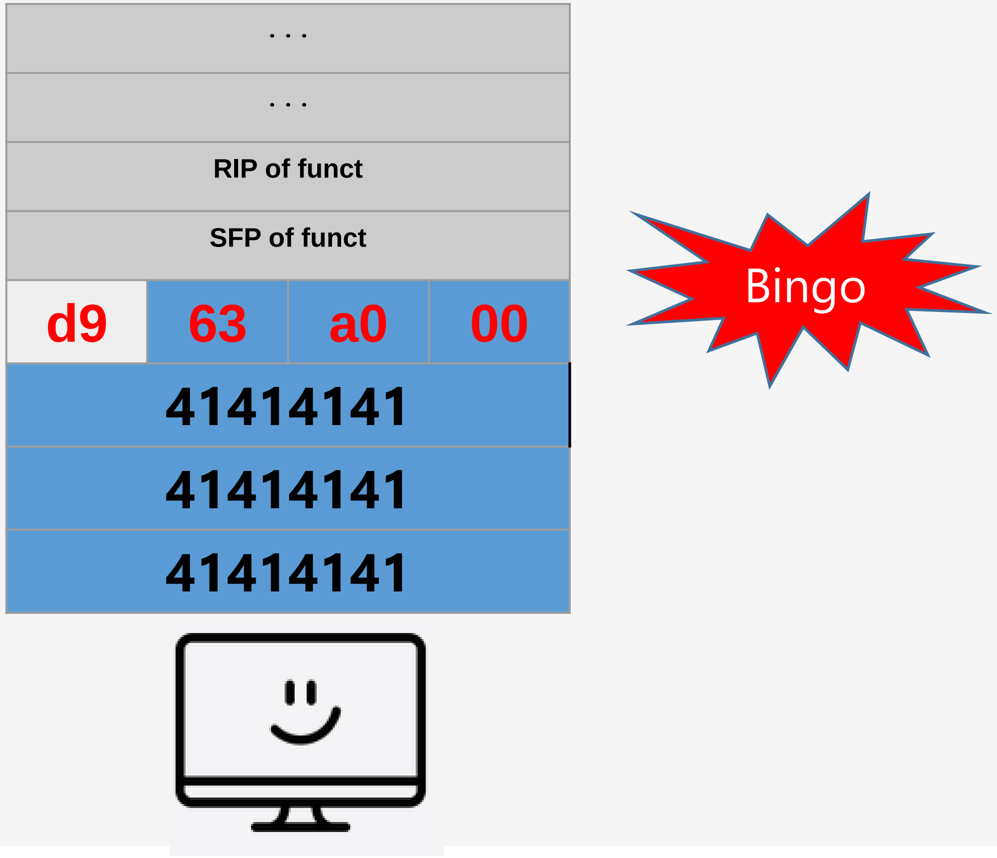
41414141

41414141

41414141

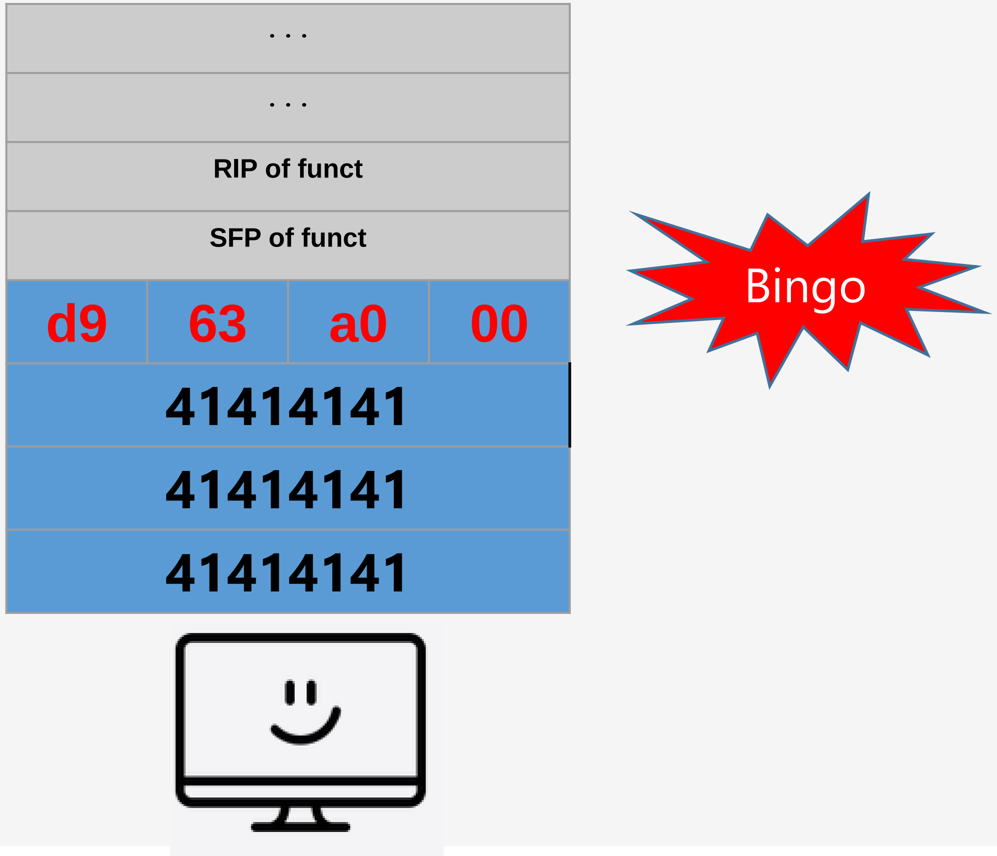
After at most 256 guesses…

Guessing canary byte by byte (32-bit)



After at most 256 guesses…

Guessing canary byte by byte (32-bit)



. . .

. . .

RIP of funct

SFP of funct

d9 63 a0 00

41414141

41414141

41414141

Canaries and Forking Servers

- Now the complete 32-bit canary is revealed

- Each byte will take at most 256 guesses

- The right most byte is always \x00, so no guess is needed (though we do show it in the animation to make it appears to be more "complete") to guess the canary byte-by-byte (max 256 tries per byte)
  - In total, at most 256+256+256 = 768 guesses for 32-bit canary!