# intro to pwn

put the hell(o kitty) in shell

joyce lai

# the plan

## 0x0 background

the stack and stuff

## 0x1 bug overview

""features""

## 0x2 real life!

boing boing

## 0x3 example

try it!

# 0x0

## background

# what is pwn?

## a category/a lifestyle

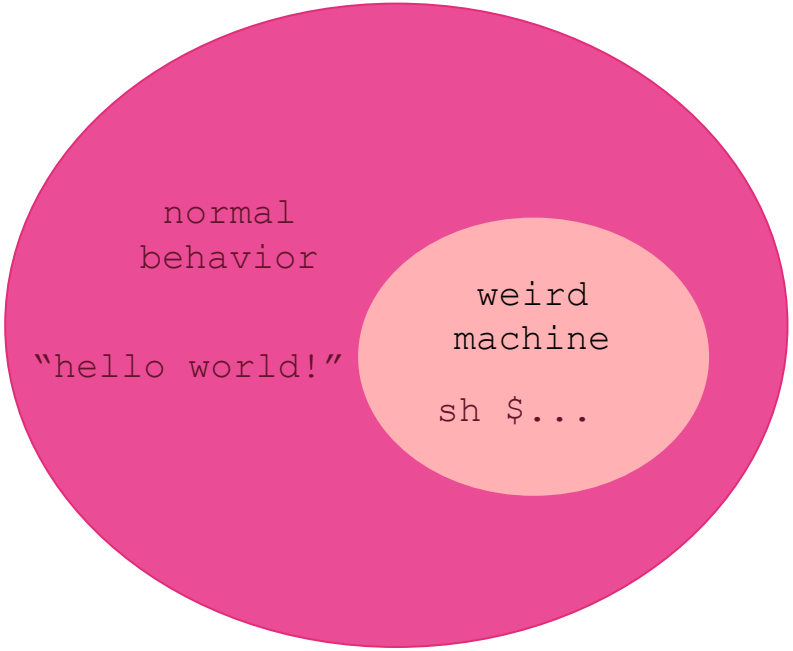
a. binary exploitation and pwn challenges in ctf
b. require you to manipulate input to a program to cause **unintended output/execution**

## vocab

**weird machine** - output/execution of a program outside of design

**payload** - data given to the program to cause unexpected behavior

# the basics



# the basics

prerequisites:

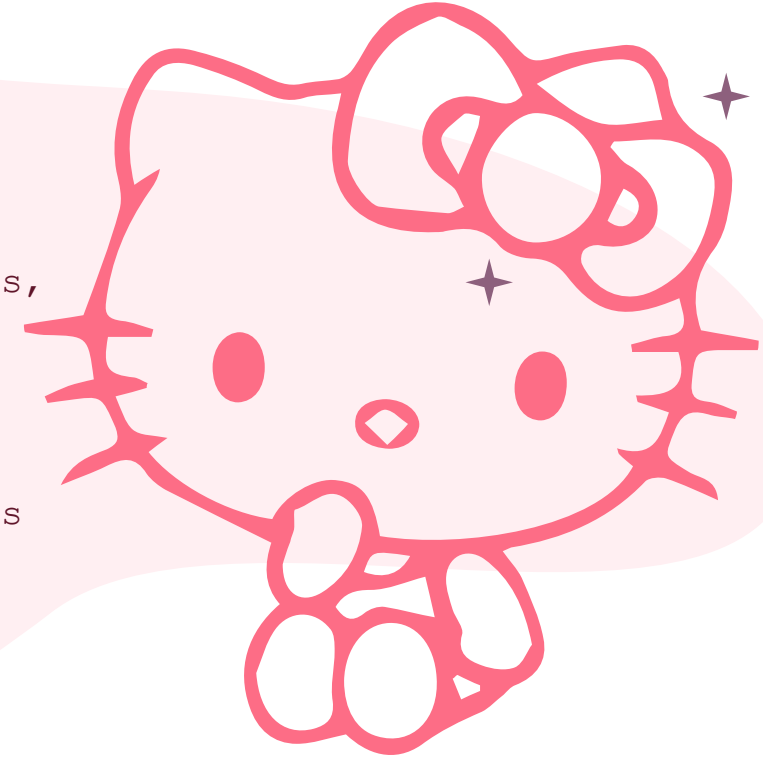
- C/C++ (very memory unsafe)

today: **x86 (32-bit)** ISA semantics

- this is only one of many many
- powerpc, arm, riscv, x8664, mips, etc.

so what is an isa?

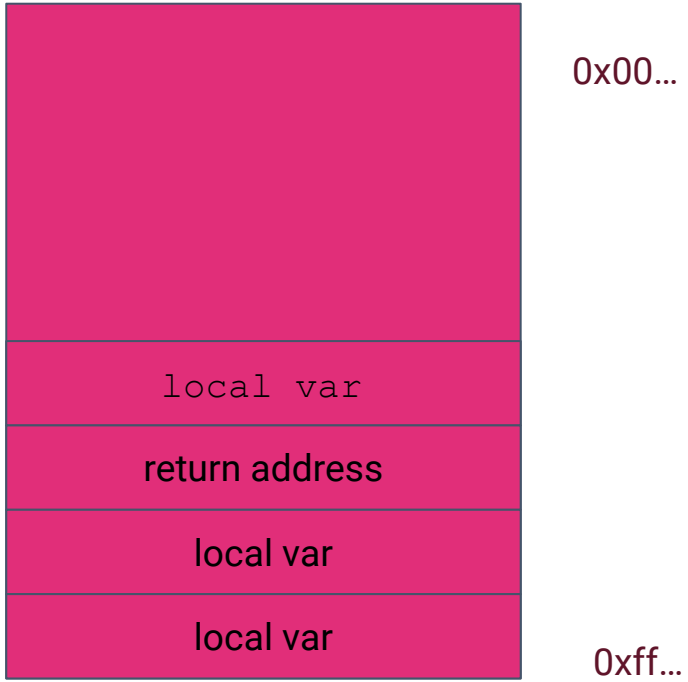
- provides an **assembly syntax**
- **convention for passing** arguments
- outlines capabilities for an architecture
- etc.



```c
#include <bits/stdc++.h>
using namespace std;

int cow(int x){
    return x + 1;
}

int main() {
    int meow = 5;
    int sheep = cow(meow);

    printf("%d\n", sheep);
}
```

```asm
cow:
        pushl %ebp
        movl %esp,%ebp
        movl 8(%ebp),%eax
        incl %eax
        jmp .L1
.L1:
        leave
        ret
.LC0:
        .byte 0x25,0x64,0xa,0x0
main:
        pushl %ebp
        movl %esp,%ebp
        subl $8,%esp
        movl $5,-4(%ebp)
        pushl -4(%ebp)
        call cow
        movl %eax,-8(%ebp)
        pushl -8(%ebp)
        pushl $.LC0
        call printf
        leave
        ret
```

# the stack




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

  int main(){
    long val=0x41414141;
    char buf[20];

    printf("Correct val's value from 0x41414141 -> 0xdeadbeef!\n");
    printf("Here is your chance: ");
    scanf("%24s",&buf);

    printf("buf: %s\n",buf);
    printf("val: 0x%08x\n",val);

    if(val==0xdeadbeef)
        system("/bin/sh");
    else {
        printf("WAY OFF!!!!\n");
        exit(1);
    }

    return 0;
  }
```

credit to overthewire challenge: narnia0

# the stack

| | |
|---|---|
| 0x41414141 | 0x00... |
| 0x41414141 | |
| 0x41414141 | |
| 0x41414141 | |
| 0x41414141 | |
| 0x41414141 | |
| 0x41414141 | |
| 0x42424242 | 0xff... |

# 0x1
# bug overview

oh a wide menagerie

**what is a bug?** in short… **unintended** behavior

you're probably unfortunately familiar:

- off by one
- logic errors
- compiler errors…
- etc.

sometimes we have bugs with **bigger consequences**

- usually in terms of user input

# specific bugs

## buffer overflow

"overflowing" the bounds of a buffer and writing to unintended memory (many overflow variants)

## logic error

integer overflow/underflow

- can be used to bypass checks

## format string

%x, %s, etc.

- combined with user input, can be used to leak program memory

# specific bugs

## use after free

after free'ing a pointer, it is reused allowing possible control

## double free

freeing a pointer multiple times corrupts memory and allows for leaks and injection

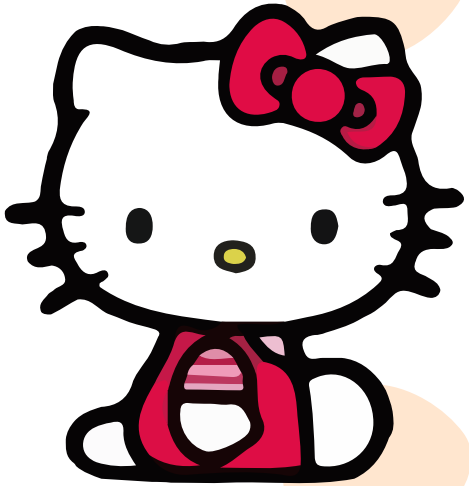
## type confusion

more JS specific

- we can use this to gain memory primitives

## we call these: primitives

```c
#include "stdio.h"
#include "stdlib.h"
#include "unistd.h"
#include "stdint.h"

void win(){
    puts("You win");
    exit(0);
}

void hello(){
    uint64_t addr=(uint64_t)&win;
    char mesg[]="hello\n\0";
    char buf[8];
    read(0, buf, 0x100);
    printf("%s", mesg);
}

void challenge3(){
    char buf[8];
    read(0, buf, 0x100);
}

int main(){
    setvbuf(stdout, 0, 2, 0);
    setvbuf(stdin, 0, 2, 0);
    hello();
    puts("challenge3");
    challenge3();
}
```



# gdb crash course

**gnu debugger.**

## program control

**c(ontinue):** run the program
**ni:** next instruction
**si:** step instruction
**n/s:** next/step code line

## display memory

**x/nsf <var/addr>:** show memory at an address or variable

- n: number to show
- s: size (dw,b,etc.)
- f: format (hex, integer)

**display/nsf <var/addr>:** for a variable you want to display every step

## breakpoints

**b <address/func/line>**
**cond <breakpoint> <condition>**

## useful misc

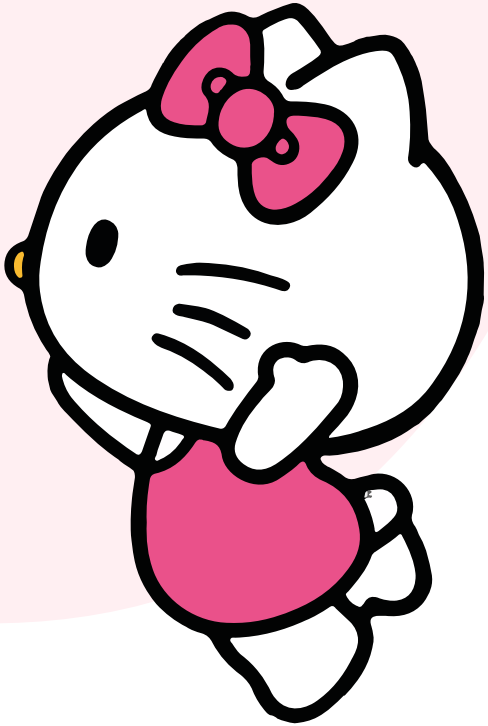
**watchpoints**
**finish:** finish the current function
**set logging file <hello_kitty.log>**
**target remote <host>:<port>**

wait, we can write **onto the stack** and we can also jump **anywhere** we want?

shellcode!

# further study...

# shellcode :3



## what?

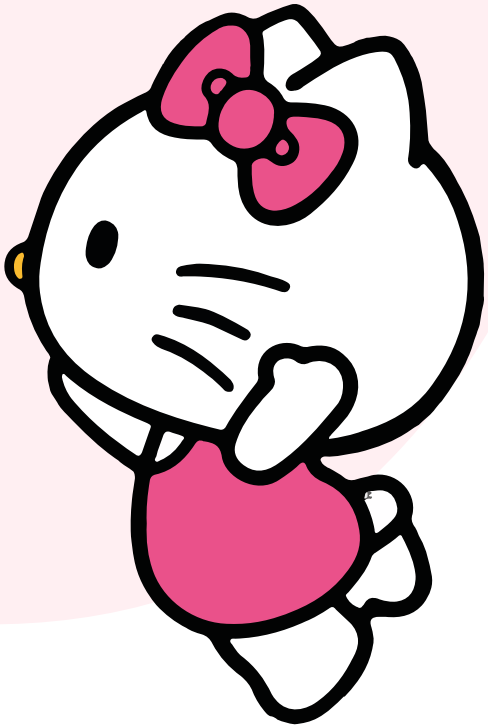
oftentimes our goal is to get a **shell** or **reverse shell** if our target is remote

- allow further control over system
- privilege of whatever program was executing

restrictions:

- nullbytes
- length
- other input limitations

# shellcode :3

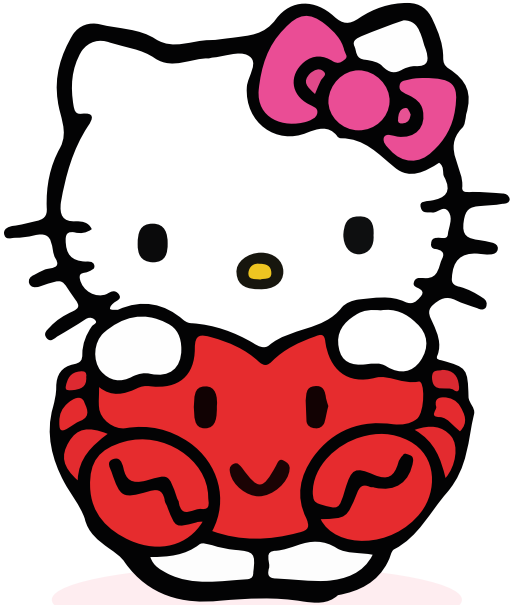


simplest form:

**exec("/bin/sh", nullptr, nullptr)**

- lots of null bytes :(((
- ways to get around this
- jumps, etc.

```
section .text
    global _start

_start:
    xor eax, eax
    push eax
    push 0x68732f2f ; push /bin//sh
    push 0x6e69622f
    mov ebx,esp
    mov ecx, eax
    mov edx, eax
    mov al, 0xb ; set sys_execve
    int 0x80
```



# 0x2

## real life!

like a box of chocos

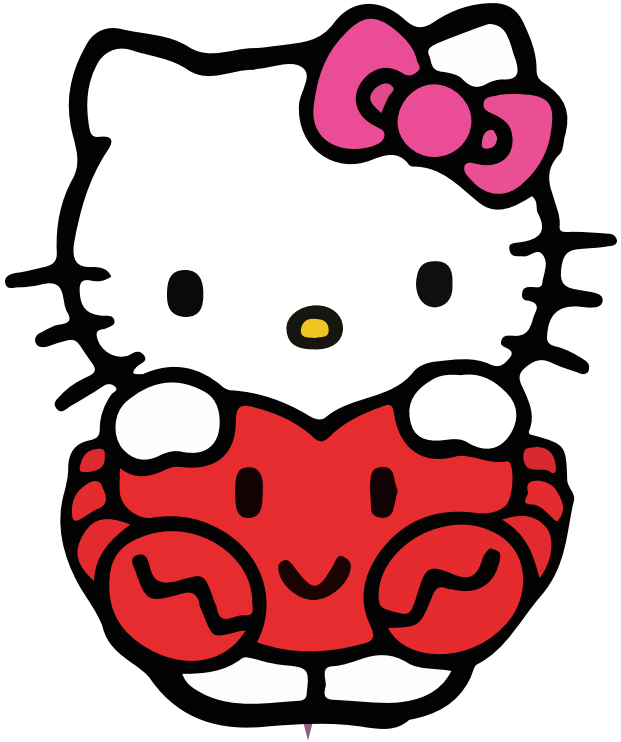
# mitigations :(

**Stack Canaries**

- added buffer overflow protection
- need to leak canary

**ASLR**

- randomize locations of libraries and code segments
- pain



**DEP**

- can no longer put shellcode on stack
- separation of permissions

**CFI**

- newer technology, prevent arbitrary jumping

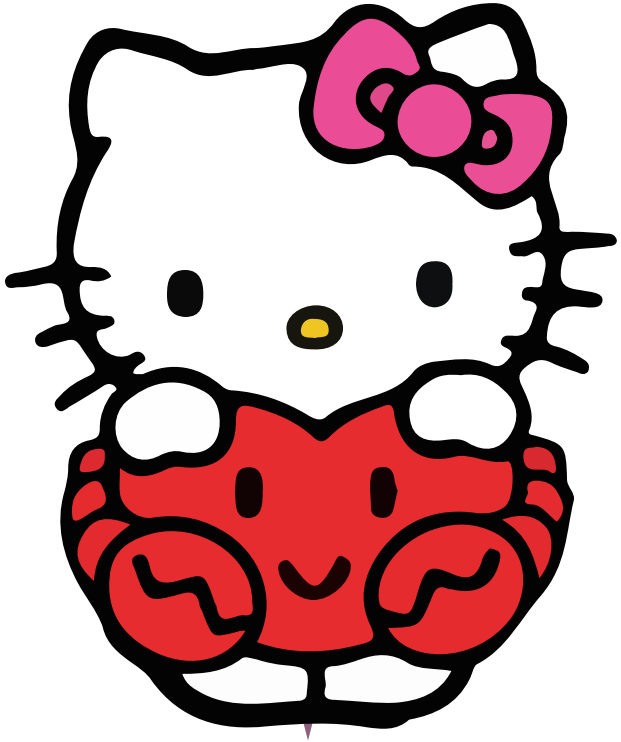
# defeats :)

**Stack Canaries**

- static canaries can be found through gdb
- printf/memory leak can show us

**ASLR**

- possible leak through plt/got tables
- it only takes one



**DEP**

- mprotect
- rop'ing

**CFI**

- very… careful rop

# cycle of pwn

## source?

sometimes, we'll have the src code to look at

## analyze

when we have the source, it can be easier to spot bugs

## get pwned

send your payload >:)

## why.

your bug was just patched/they just came out with new cfi/aslr

infinite cycle of torment

## rev time

another time another talk

# 0x3

## example time

your turn! get food first tho :3

# Thanks!