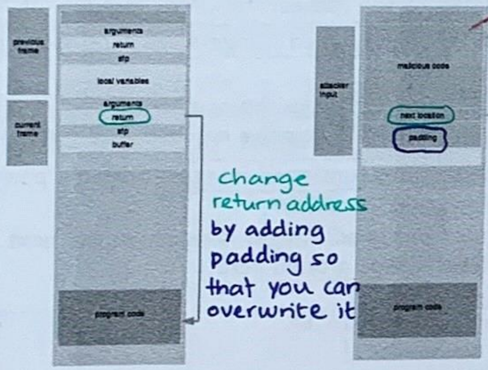
16

BUFFER OVERRUN ATTACKS

Control hijacking



change return address by adding padding so that you can overwrite it

make your malicious code be part of your maliciously provided input so tht is written in the address space of the process tht. is running

A buffer overflow can change the flow of execution of the program:

1) load malicious code into memory
2) make %eip point to it

Shellcode injection

Goal: "spawn a shell" - will give the attacker general access to the system

```
#include stdio.h
void main() {
char *name[2];
name[0] = "/bin/sh";
name[1] = NULL;
execve(name[0], name, NULL);
}
```

C code

```
"\x31\xc0"
"\x50"
"\x68" "//sh"
"\x68" "/bin"
"\x89\xe3"
"\x50"
...
```

Machine code (part of attacker's input)

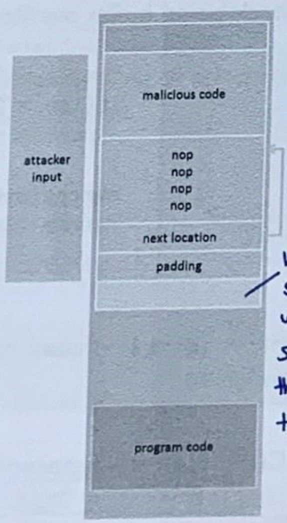
[Low-level]

- must inject the machine code instructions (code ready to run)
- the code cannot contain any zero bytes (printf, gets, strcpy will stop copying)
  (Null characters) \0
- can't use the loader (we're injecting)

The return address

Challenge: find the address of the injected malicious code?

- If code accessible: we know how far is the overflowed variable from the saved %ebp
- If code not accessible: try different possibilities! In a 32 bits memory space, there are 2^{32} possibilities



We need to know the size of the buffer we are overflowing s.t. we know exactly the padding we need to hit the %ebp + 8 position

- guess approximate stack state when the function is called
- insert many NOPs before Shell Code

NOP Sled — technique to get the return address

is a sequence of NOP instructions meant to 'slide' the CPU's instr. execution flow to its final desired destination

Solves the problem of finding exact address of the buffer by increasing size of target area.

Unsafe libc functions

```
strcpy (char *dest, const char *src)

strcat (char *dest, const char *src)

gets (char *s)

scanf (const char *format, ...)

...
```

Do not check bounds of buffers they manipulate!!

These fns do not check the size of the buffer they write to.

SOLN:

Use safe functions

```
int read_stdi(void){
   char buf[128];
   int i;
   fgets(buf, sizeof(buf), stdin);
   i = atoi(buf);
   return i;
}
```

But then...

... your program is as secure as its programmer is cautious. It is now up to the programmer to include all the necessary checks in his program :-/ and this is a tricky one...

Arithmetic overflows

- Limitation related to the representation of integers in memory
- In 32 bits architectures, signed integers are expresses in two's compliment notation
  - 0x00000000 - 0x7fffffff: positive numbers 0 - $(2^{31} - 1)$
  - 0x80000000 - 0xffffffff: negative numbers $(-2^{31} + 1)$ - (-1)
- In 32 bits architectures, unsigned integers are only positive numbers 0x00000000 - 0xffffffff.
  Once the highest unsigned integer is reached, the next sequential integer wraps around zero.

```
# include <stdio.h>
int main(void){
  unsigned int num = 0xffffffff;
  printf(''num + 1 = 0x%x\n'', num + 1);
  return 0;
}
```

The output of this program is: num + 1 = 0x0

Arithmetic overflow exploit (1)

- Stack-based buffer overflow due to arithmetic overflow:

```
int catvars(char *buf1, char *buf2,
                 unsigned int len1, unsigned int len2){
  char mybuf[256]; → allocates local buffer of 256 bytes
  if((len1 + len2) > 256){
    return -1;
  }
  memcpy(mybuf, buf1, len1);
  memcpy(mybuf + len1, buf2, len2);
  do_some_stuff(mybuf);
  return 0;
}
```

len1 = 259
len2 = v. large num

Check can be bypassed by using suitable values for len1 **and** len2, *e.g.* len1 = 0x00000103, len2 = 0xfffffffc, len1+len2 = 0x000000ff (decimal 255)

Due to wrap-around!

Arithmetic overflow exploit (2)

- Stack-based buffer overflow due to arithmetic overflow:

```
int catvars(char *buf, int len){
  char mybuf[256];
  if(len > 256){
    return -1;
  }
  memcpy(mybuf, buf, len);
  return 0;
}
```

The memcpy fn takes unsigned int!

```
memcpy(void *s1, const void *s2, size_t n); // size_t
is unsigned
```

Check can be bypassed by using suitable values for len, *e.g.* len = -1 = 0xffffffff, will be interpreted as an unsigned integer encoding the value $2^{32} - 1$!

Arithmetic overflow exploit (3)

- Heap-based buffer overflow due to arithmetic overflow:
  - Memory **dynamically** allocated will persist across multiple function calls.
  - This memory is allocated on the **heap** segment.
  - Heap-based buffer overflows are more complex, and require understanding garbage collection and heap implementation.

```
int myfunction(int *array, int len){
  int *myarray, i;
  myarray = malloc(len * sizeof(int));
  if(myarray == NULL){
    return -1;
  }
  for(i = 0; i < len; i++){
    myarray[i] = array[i];
  }
  return myarray;
}
```

Can allocate a size 0 buffer for `myarray` by using suitable value for `len`: $\boxed{\texttt{len = 1073741824}}$, $\boxed{\texttt{sizeof(int) = 4}}$, In a 32-bit machine

$\boxed{\texttt{len*sizeof(int) = 0}}$

$1073741824 \times 4 = 4294967296$

$2^{32} = 4294967296$

Hence it will wrap around to zero!

Format strings

[Ref] scut/team teso. Exploiting Format String Vulnerabilities

- A format function takes a variable number of arguments, from which one is the so called format string

  Examples: `fprintf`, `printf`, ..., `syslog`, ...

  `printf(''The amount is %d pounds\n'', amnt);`

- The behaviour of the format function is controlled by the format string. The function retrieves the parameters requested by the format string from the stack

  Example: `printf(fmt_str, arg`$_1$`, ..., arg`$_n$`);`

...
arg_n
...
arg_1
&fmt_str
ret
sfp
...

Exploiting format strings

- If an attacker is able to provide the format string to a format function, a format string vulnerability is present

Does not specify a format string, allowing the user to supply one

```
int vulnerable_print(char *user) {
  printf(user);
}

int safe_print(char *user){
  printf ("%s", user);
}
```

will be interpreted as a format string

Format strings exploits

This instructs the printf fn to retrieve 5 params from the stack and display them as 8-digit padded hex numbers.

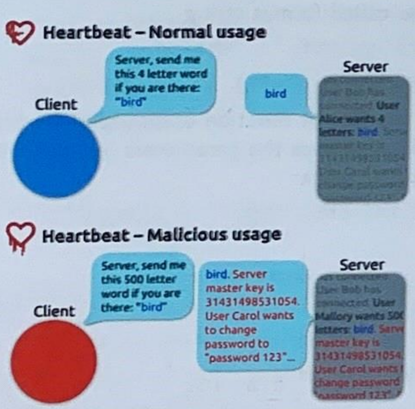
1) We can view the stack memory at any location
   - walk up stack until target pointer found
   - `printf (''%08x.%08x.%08x.%08x.%08x|%s|'');`
   - A vulnerable program could leak information such as passwords, sessions, or crypto keys

2) We can write to any memory location
   - `printf(''hello %n'', &temp)` – writes '6' into temp
   - `printf(''hello%08x.%08x.%08x.%08x.%n'')`

%n – allows you to print the no. of chars you've printed on the std. output

&temp – is the address where it'll be printed

TLS Heartbleed



→ BUFFER OVERRUN

TLS Heartbleed

Then, OpenSSL will uncomplainingly copy 65535 bytes from your request packet, even though you didn't send across that many bytes:

```
/* Allocate memory for the response, size is 1 byte
 * message type, plus 2 bytes payload length, plus
 * payload, plus padding
 */
buffer = OPENSSL_malloc(1 + 2 + payload + padding);
bp = buffer;

/* Enter response type, length and copy payload */
*bp++ = TLS1_HB_RESPONSE;
s2n(payload, bp);
memcpy(bp, pl, payload);  - No checks
bp += payload;
/* Random padding */
RAND_pseudo_bytes(bp, padding);

r = dtls1_write_bytes(s, TLS1_RT_HEARTBEAT, buffer, 3 + payload +
```

That means OpenSSL runs off the end of your data and scoops up whatever else is next to it in memory at the other end of the connection, for a potential data leakage of approximately 64KB each time you send a malformed heartbeat request.