Other memory exploits

Other attacks

- The code injection attack we have just considered is called **stack smashing**
  - The term was coined by *Aleph One* in 1996
- Constitutes an **integrity violation**, and arguably a **violation of availability**
- Other attacks exploit bugs with buffers, too

Heap overflow

- Stack smashing overflows a stack allocated buffer
- You can also **overflow a buffer** allocated by `malloc`, which resides on the **heap**

Heap overflow

```
typedef struct _vulnerable_struct {
  char buff[MAX_LEN];
  int (*cmp)(char*,char*);
} vulnerable;

int foo(vulnerable* s, char* one, char* two)
{
 strcpy( s->buff, one );
 strcat( s->buff, two );
 return s->cmp( s->buff, "file://foobar" );
}
```

copy ***one*** *into* ***buff***

copy ***two*** *into* ***buff***

must have ***strlen(one)+strlen(two) < MAX_LEN***

or we overwrite s->cmp

Heap overflow variants

- **Overflow into the C++ object *vtable***
  - C++ objects (that contain virtual functions) are represented using a *vtable*, which contains pointers to the object's methods
  - This table is analogous to `s->cmp` in our previous example, and a similar sort of attack will work
- **Overflow into adjacent objects**
  - Where buff is not collocated with a function pointer, but is allocated near one on the heap
- **Overflow heap metadata**
  - Hidden header just before the pointer returned by malloc
  - Flow into that header to corrupt the heap itself
    - Malloc implementation to do your dirty work for you!

Integer overflow

```
void vulnerable()
{
  char **response;
  int nresp = packet_get_int();
  if (nresp > 0) {
   response = malloc(nresp*sizeof(char*));
   for (i = 0; i < nresp; i++)
    response[i] = packet_get_string(NULL);
  }
```

HUGE

Wrap-around

Overflow

- If we set **`nresp`** to 1073741824 and **`sizeof(char*)`** is 4
- then **`nresp*sizeof(char*)`** overflows to become 0
- subsequent writes to allocated **`response`** overflow it

Corrupting data

- The attacks we have shown so far **affect code**
  - *Return addresses* and *function pointers*
- But attackers can **overflow data** as well, to
  - **Modify a secret key** to be one known to the attacker, to be able to decrypt future intercepted messages
  - **Modify state variables** to bypass authorization checks (earlier example with `authenticated` flag)
  - **Modify interpreted strings** used as part of commands
    - E.g., to facilitate SQL injection, discussed later in the course

Read overflow

- Rather than permitting writing past the end of a buffer, a bug could permit **reading past the end**
- Might **leak secret information**

Read overflow

```
int main() {
  char buf[100], *p;
  int i, len;
  while (1) {
   p = fgets(buf,sizeof(buf),stdin);
   if (p == NULL) return 0;
   len = atoi(p);
   p = fgets(buf,sizeof(buf),stdin);
   if (p == NULL) return 0;
   for (i=0; i<len; i++)
    if (!iscntrl(buf[i])) putchar(buf[i]);
    else putchar('.');
   printf("\n");
 }}
```

May exceed actual message length!

Read integer

Read message

Echo back (partial) message

Sample transcript

```
% ./echo-server
24
every good boy does fine
ECHO: |every good boy does fine|
10
hello there
ECHO: |hello ther|
25
hello
ECHO: |hello..here..y does fine.|
```

OK: input length < buffer size

BAD: length > size !

leaked data

Heartbleed

- The **Heartbleed bug** was a read overflow in exactly this style
- The SSL server should accept a “heartbeat” message that it echoes back
- The heartbeat message specifies the length of its echo-back portion, but the buggy SSL **software did not check the length was accurate**
- Thus, an attacker could request a longer length, and **read past the contents of the buffer**
  - Leaking passwords, crypto keys, …

Stale memory

- A **dangling pointer bug** occurs when a pointer is freed, but the program continues to use it
- An attacker can **arrange for the freed memory to be reallocated** and under his control
  - When the dangling pointer is dereferenced, it will access attacker-controlled data

```
struct foo { int (*cmp)(char*,char*); };

struct foo *p = malloc(…);
free(p);
…
q = malloc(…) //reuses memory
*q = 0xdeadbeef; //attacker control
…
p->cmp("hello","hello"); //dangling ptr
```

IE's Role in the Google-China War

By Richard Adhikari
TechNewsWorld
01/15/10 12:25 PM PT

The hack attack on Google that set off the company's ongoing standoff with China appears to have come through a zero-day flaw in Microsoft's Internet Explorer browser. Microsoft has released a security advisory, and researchers are hard at work studying the exploit. The attack appears to consist of several files, each a different piece of malware.

Computer security companies are scurrying to cope with the fallout f et Explorer (IE) flaw that led to cyberattacks on Google and its corporat s.

The zero-day attack that exploited IE is part are that is keeping researchers very busy.

"We're discovering th ate basis, and we've seen about a dozen files dropped o Alperovitch, vice president of research at McAfee Labs, told Tech

The attack on Google, which appeared to originate in China, have sparked a feud between the Internet giant and the nation's government over censorship, and it could result in Google pulling away from its business dealings in the country.

Pointing to the Flaw

The vulnerability in IE is an invalid pointer reference, Microsoft said in security advisory 979352, which it issued on Thursday. Under certain conditions, the invalid pointer can be accessed after an object is deleted, the advisory states. In specially crafted attacks, like the ones launched against Google and its customers, IE can allow remote execution of code when the flaw is exploited.

Dangling pointer dereference!