Memory safety, continued

With material from Mike Hicks and Dave Levin

Today’s agenda

• Finishing up shellcodes
• gdb tutorial
• Other memory exploits
• Defenses and the continuing arms race
  • Start today, continue on Thursday
Stack and functions: Summary

**Calling function:**
1. **Push arguments** onto the stack (in reverse)
2. **Push the return address**, i.e., the address of the instruction you want run after control returns to you
3. **Jump** to the function’s address

**Called function:**
4. **Push the old frame pointer** onto the stack: `%ebp`
5. **Set frame pointer** to where the end of the stack is right now: `%ebp = %esp`
6. **Push local variables** onto the stack

**Returning from function:**
7. **Reset the previous stack frame**: `%esp = %ebp, pop %ebp`
8. **Jump back** to return address: `pop %eip`
Challenge 3

Finding the return address

- We need an address to point to: Where is our shellcode?

- One approach: try a lot of different values!
  - Worst case scenario: it’s a 32 (or 64) bit memory space, which means $2^{32}$ ($2^{64}$) possible answers

- Without address randomization (discussed later):
  - Stack *always* starts from the same *fixed address*
  - Stack will grow, but usually it *doesn’t grow very deeply* (unless the code is heavily recursive)
Improving our chances: **nop** sleds

**nop** is a single-byte no-op instruction (just moves to the next instruction)

Now we improve our chances of guessing by a factor of \#nops
Putting it all together

Fill in the space between the target buffer and the %eip to overwrite

%eip

padding

good guess

buffer

nop sled

malicious code

Text ...

0xbdf

nop nop nop ...

\x0f \x3c \x2f ...

...
gdb tutorial

LIVE DEMO

http://original.livestream.com/filestore/logos/9aa0a959-2380-7191-77e8-06fdaf
Some gdb commands

```
if
```
Show info about the current frame (prev. frame, locals/args, %ebp/%eip)

```
ir
```
Show info about registers (%eip, %ebp, %esp, etc.)

```
x/<n> <addr>
```
Examine <n> bytes of memory starting at address <addr>

```
b <function>
```
Set a breakpoint at <function>

```
s
```
Step through execution (into calls)
• run, breakpoint, step, next, continue
  • stepi, nexti
• print, x
• backtrace, info [frame, registers, args, locals]
• list, disas
inside factorial(1)

(gdb) x/32xw $ebp

0xbffff8a8: 0xbffff8c8 0x00001eee 0x00000001 0x00000007
0xbffff8b8: 0x00000002 0x00000001 0x00000002 0xbfffff9cc
0xbffff8c8: 0xbffff8e8 0x00001eee 0x00000002 0x00000000
0xbffff8d8: 0x00000003 0x00000001 0x00000003 0x00000000
0xbffff8e8: 0xbffff918 0x00001f46 0x00000003 0x00000000
0xbffff8f8: 0x00000000 0x00000000 0x00001f0c 0xa55aa55a
0xbffff908: 0x00000003 0xbffff948 0x00000002 0x00000000
0xbffff918: 0xbffff940 0x9be446d9 0x00000002 0xbfffff948

Saved base pointer  Argument
Saved instruction pointer  Local
Other memory exploits
Heap overflow

• Stack smashing overflows a stack-allocated buffer
• You can also **overflow a buffer** allocated by `malloc`, which resides on the **heap**
  • What data gets overwritten?
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 );  \textit{copy one into buff} 
    strcat( s->buff, two );  \textit{copy two into buff} 
    return s->cmp( s->buff, "file://foobar" );
}

\textit{must have} \texttt{strlen(one)+strlen(two) < MAX_LEN}  \textit{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!
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);
    }
}

• What if we set \texttt{nresp} = 1073741824?
• Assume \texttt{sizeof(char*)} = 4
• How many bytes are \texttt{malloc’d}?

• The \texttt{for} loop now creates an overflow!
Corrupting data

- Attacks so far primarily 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

```c
int main() {
    char buf[100], *p;

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

- Read integer
- Read message
- Echo back (partial) message

`len` may exceed actual message length!
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.|
Heartbleed, again
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.
Stale memory

```
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
```

- When the dangling pointer is dereferenced, it will access attacker-controlled data
The vulnerability in IE is an invalid pointer reference, Microsoft said in a 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.

Computer security companies have been trying to cope with the fallout from the Internet Explorer (IE) flaw that led to this week's attack on Google and its corporate and individual customers.

The advisory states that an unexploited IE is part of a lethal cocktail of malware that is keeping security experts very busy.

"We're discovering things on an up-to-the-minute basis, and we've seen about a dozen files dropped on infected PCs so far," Dmitri Alperovitch, vice president of research at McAfee Labs, told TechNewsWorld.

The attacks 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 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.
Format string vulnerabilities
Formatted I/O

- Recall: C’s `printf` family of functions
- Format specifiers, list of arguments
  - Specifier indicates type of argument (%s, %i, etc.)
  - Position in string indicates argument to print

```c
void print_record(int age, char *name)
{
    printf("Name: %s\tAge: %d\n", name, age);
}
```
What’s the difference?

```c
void vulnerable()
{
    char buf[80];
    if(fgets(buf, sizeof(buf), stdin)==NULL)
        return;
    printf(buf);  \textbf{Attacker controls the format string}
}

void safe()
{
    char buf[80];
    if(fgets(buf, sizeof(buf), stdin)==NULL)
        return;
    printf("%s", buf);
}
```
printf implementation

int i = 10;
printf("%d %p\n", i, &i);

• printf takes a variable number of arguments
• Doesn’t know where the stack frame “ends”
• Keeps reading from stack until out of format specifiers
```c
void vulnerable()
{
    char buf[80];
    if(fgets(buf, sizeof(buf), stdin)==NULL)
        return;
    printf(buf);
}
```

"%d %x"

caller’s stack frame
Format string vulnerabilities

- `printf("100% dinosaur");`
  - Prints stack entry 4 bytes above saved `%eip`

- `printf("%s");`
  - Prints bytes *pointed to* by that stack entry

- `printf("%d %d %d %d ...");`
  - Prints a series of stack entries as integers

- `printf("%08x %08x %08x %08x ...");`
  - Same, but nicely formatted hex

- `printf("100% not vulnerable!")`
  - *WRITES* the number 3 to address pointed to by stack entry
Why is this a buffer overflow?

- We should think of this as a buffer overflow in the sense that
  - The stack itself can be viewed as a kind of buffer
  - Size of that buffer is determined by the number and size of the arguments passed to a function
- Providing a bogus format string thus induces the program to overflow that “buffer”
Vulnerability prevalence

number of vulnerabilities that involve format string bugs

100% preventable!

Time to switch hats

We have seen many styles of attack

How can we defend against them?
Defenses
Against memory and buffer attacks

http://www.full-stop.net/wp-content/uploads/2012/05/Great-wall-of-china.jpeg
Stepping back

What do these attacks have in common?

1. The attacker is able to control some data that is used by the program

2. The use of that data permits unintentional access to some memory area in the program
   • Past a buffer
   • To arbitrary positions on the stack / in the heap
Outline

- **Memory safety and type safety**
  - Properties that, if satisfied, ensure an application is immune to memory attacks
- Automatic defenses
  - Stack canaries
  - Address space layout randomization (ASLR)
- Return-oriented programming (ROP) attack
  - How Control Flow Integrity (CFI) can defeat it
- Secure coding
“Once you learn, though, you’ll never forget.”

Memory Safety
What is memory safety?

A memory safe program execution:

1. Only creates pointers through **standard means**
   - \( p = \text{malloc}(\ldots), \) or \( p = \&x, \) or \( p = \&\text{buf}[5], \) etc.

2. Only uses a pointer to access memory that **“belongs” to that pointer**

Combines two ideas:

- **temporal safety** and **spatial safety**
Spatial safety

• View pointers as **capabilities**: triples \((p,b,e)\)
  • \(p\) is the actual pointer (current address)
  • \(b\) is the base of the memory region it may access
  • \(e\) is the extent (bounds) of that region (count)

• **Access allowed** \(iff\) \(b \leq p \leq (e-\text{sizeof}({\text{typeof}(p)}))\)

• Operations:
  • Pointer arithmetic increments \(p\), leaves \(b\) and \(e\) alone
  • Using &: \(e\) determined by size of original type
Examples

```c
int x;        // assume sizeof(int)=4
int *y = &x;  // p = &x, b = &x, e = &x+4
int *z = y+1; // p = &x+4, b = &x, e = &x+4
*y = 3;       // OK: &x ≤ &x ≤ (&x+4)−4
*z = 3;       // Bad: &x ≤ &x+4 ≤ (&x+4)−4
```

```c
struct foo f = { "cat", 5 };  
struct foo { 
    char buf[4];
    int x;
};
```

```c
char *y = &f.buf; // p = b = &f.buf, e = &f.buf+4
y[3] = 's';  // OK: p = &f.buf+3 ≤ (&f.buf+4)−1
y[4] = 'y';  // Bad: p = &f.buf+4 ≤ (&f.buf+4)−1
```
struct foo {
    int x;
    int y;
    char *pc;
};
struct foo *pf = malloc(...);
pf->x = 5;
pf->y = 256;
pf->pc = "before";
pf->pc += 3;
int *px = &pf->x;