

Memory safety

- The concept of memory safety is one closely related to that of type safety
- For a programming language to be memory safe, writes to a given type must not exceed the bounds of that type
- Again, Java is memory safe, C is not

Where are we?

- We have built up an understanding of both process organisation in memory and in particular the role of the stack in program execution
- As such, we are ready to take a first look at some of the more common vulnerabilities introduced through C coding error
- We do not yet investigate in detail how these vulnerabilities can be exploited: we limit ourselves to understanding them and outlining how, in principle, they could be exploited
- Exploit development and implementation is covered in detail in upcoming lectures

- Each of the vulnerabilities covered is dealt with in detail in the standard secure coding references
- Seacord covers buffer overflow and off-by-one errors in Chapter 2, “Strings” (he also covers arc injection which we’ll cover later)
- He covers format string attacks (at length) in Chapter 6, “Formatted Output”
- Other relevant links are provided off the course home page

Vulnerabilities

- Buffer overflows
- Off-by-one errors
- Format string attacks

Buffer overflows

- Careless use or a misunderstanding of C's many string handling (although the problem extends to data copying functions in general) may leave an application vulnerable to attack
- In the following C excerpt a character array, `dest` is allocated on the stack
- The `strcpy` function is used to copy the user-supplied `src` string into `dest`
- What if the `src` string is longer than the size of the `dest` array?

```
void
foo(char *src)
{
    char dest[64];

    strcpy(dest, src);
}
```

Stack



+04	Return address
+00	EBP
-64	Local buffer

- Notice that the size of the `dest` array is not passed as a parameter to `strcpy`
- Obviously, the size of `dest` is in no way taken into account in the implementation of `strcpy`
- `strcpy`, being oblivious to the size of `dest`, will copy from `src` until it reaches the end of string character: `'\0'`
- When `src` is bigger than `dest` and once `dest` has been filled, `strcpy` will continue to write beyond its bounds and `dest` overflows: we have a classic buffer overflow

- The overflow must go somewhere, where does it go?
- `strcpy` writes from low to high memory addresses and so will start at the “bottom” of `dest` and work its way up the stack overwriting whatever lies along the way
- Crucially, not far above `dest` on the stack lies the stored return address: it can be overwritten during an overflow
- Not only can the return address be overwritten but it can be purposefully rewritten to point to a new location

- Where will the new return address point to? Usually somewhere inside `src` itself where the attacker has placed executable code
- Injected attack code (the “payload”) is usually padded with `NOP` instructions to widen the landing area for a guessed return address
- A padded landing area may also help circumvent defensive techniques such as stack randomisation
- Sometimes the code which an attacker aims to execute may already exist in the process address space and need not be injected at all (or we may seek to skip a block of privilege-checking code)

- Traditional buffer overflows are stack-based but they need not be
- In the following C excerpt `strcpy` is used to write to a BSS-based character array
- Above the array, in the BSS section of the process image, lies a function pointer
- By overflowing the array an attacker can reach the function pointer and rewrite it to point at some other function

```
char dest[64];
void (*fptr)(void);

void
foo(char *src)
{
    fptr = bar;

    strcpy(dest, src);

    (*fptr)();
}
```

```
void
bar(void)
{
    printf("`Hi\n`");
}
```

BSS



+00	Function ptr
-64	BSS buffer

Off-by-one errors

- An off-by-one error allows an attacker to overflow a buffer by one byte and in so doing to possibly overwrite the least significant byte of the calling function's EBP
- If the attacker is lucky the calling function's now modified EBP may point into an area under her control e.g. the overflowing buffer itself
- The attacker might try filling the buffer with the address of some code she would like to execute
- What happens when the overflow occurs?

```
void
foo(char *src)
{
    char dest[64];

    strncpy(dest,
            src,
            sizeof (dest));

    dest[sizeof (dest)] =
        '\0';
}
```

Stack



+04	Return address
+00	EBP
-64	Local buffer

- Disassembling `foo`, it ends with:

```
leave [mov ebp, esp; pop ebp]
ret   [pop eip]
```

- Similarly the calling function ends with:

```
leave [mov ebp, esp; pop ebp]
ret   [pop eip]
```

- On leaving the `foo` function the calling function's now corrupted EBP is popped and execution continues in the calling function
- On leaving the calling function the stack pointer ESP is set to the corrupted EBP (`mov ebp, esp`)

- But the corrupted EBP points into the buffer that the attacker filled with the address of the payload
- EBP is popped (`pop ebp`) as usual
- EIP is popped (`ret`) and execution jumps to the payload
- The attacker must be lucky in two ways:
 1. The overflowing buffer must be immediately adjacent to EBP
 2. The corrupted EBP must point to a location whose contents are under the attacker's control

Format string attacks

- Looking at the man page, `snprintf` has the following prototype:

```
int snprintf(char *s, size_t n,  
             const char *format, /* args */);
```

- So `snprintf` takes a variable number of parameters and for each conversion specifier in `*format`, `snprintf` expects to find a corresponding argument on the stack
- By passing only conversion specifiers to `snprintf` (without any corresponding arguments) we can examine stack contents

- `snprintf` will assume its arguments to be whatever it finds on the stack
- When used properly we see something like:

```
snprintf(name, sizeof (name), ``%s %s'',  
         first, last);
```

- Can we do anything more exciting than examine stack contents?
- Going back to the man pages we find the following description of the `%n` specifier: the argument must be a pointer to an integer into which is written the number of bytes written to the output standard I/O stream so far by this call to one of the `printf` functions

- A call to one of the `printf` functions can write to memory!
- Looking at some vulnerable source code we might see the following line of code:

```
snprintf(buf, sizeof (buf), argv[1]);
```

- Our programmer has not specified a format string - she is expecting the user to enter a simple string
- By specifying our own format string in `argv[1]` we can at the very least examine stack contents and using the `%n` specifier we can even rewrite arbitrary memory locations

```
int
main(int argc, char **argv)
{
    char buf[64];
    int x;

    x = 1;

    snprintf(buf, sizeof (buf), argv[1]);

    buf[sizeof (buf) - 1] = '\\0';

    return (0);
}
```


+24	buf
+20	x
+16	argv[1]
+12	sizeof(buf)
+08	&buf
+04	Return address
+00	Caller's EBP