Introduction to Software Security

Dr Sana Belguith

Memory Corruption bugs

- Software has bugs: Sometimes we can use these bugs to violate security principles

- Memory corruption bugs (and memory safety)
  - An example of programming errors
  - Property of specific type of languages, called memory unsafe languages.
  - One of the most used bugs for exploitation: Corrupt the memory of a program to violate security principles

  Can lead to:
    - arbitrary read
    - arbitrary write
    - control flow hijack
    - control flow corruption

Pointers

- Pointers allow you to refer to (semi) arbitrary memory addresses in most programming languages
  - in C
- Some languages claim not to have them (e.g. Java)
  - Not strictly true… just not usually as easy to abuse as Cs
  - Objects
- To introduce a bug…
  - Get a pointer pointing somewhere it should not

Memory Safety

- When object boundary access is violated via pointers.
- Spatial Safety[1]:
  - A spatial safety violation is an error in which a pointer is used to access the data at a location in memory that is outside the bounds of an allocated object.
  - The error is ‘spatial’ in the sense that the dereferenced pointer refers to an incorrect location in memory.

[1]: MemSafe: ensuring the spatial and temporal memory safety of C at runtime

Memory Safety

- Temporal Safety[1]

  - A temporal safety violation is an error in which a pointer is used in an attempt to access or deallocate an object that has already been deallocated.
    - The violation is ‘temporal’ in the sense that the pointer use occurs at an invalid instance during the execution of the program (i.e., after the object to which it refers has been deallocated).

[1]: MemSafe: ensuring the spatial and temporal memory safety of C at runtime

Memory Corruption bugs

- What happens if you go beyond a local array’s end?
  - You can get arbitrary execution…
  - Sometimes called a *spatial error*
  - For further reading: See *Smashing the stack for fun and profit*

- What happens if you use memory after you’ve freed it?
  - You can get an arbitrary write…
  - Sometimes called a *temporal error*
  - For further reading: See the *malloc maleficarum*

Example

```c
#include <stdio.h>
int main(void) {
  int x;
  int buffer[4];

  x = 0;
  printf("x = %d\n", x);
  buffer[-2] = 1;
  printf("&x       = %p\n", &x);
  printf("buffer   = %p\n", buffer);
  printf("buffer-2 = %p\n", buffer-2);
  printf("x = %d\n", x);
  return 0;
}
```

Example

What happens when we compile it ?
Obviously, a warning…

```
cc  -I/opt/homebrew/opt/openjdk/include   example.c   -o example
example.c:8:3: warning: array index -2 is before the beginning of the array [-Warray-bounds]
  buffer[-2] = 1;
  ^      ~~
example.c:4:3: note: array 'buffer' declared here
  int buffer[4];
  ^
1 warning generated.
```

This is technically allowed by the C standard so not an error

Example

What happens if we run it ?
We have over-written X without explicitly pointing to it

```
% ./example
 x = 0
 &x        = 0x16bcff6e0
 buffer    = 0x16bcff6e8
 buffer-2 = 0x16bcff6e0
 x = 1
```

The problem with C (and C++)

- C was designed to write operating systems
- Programmers were expected to know what they were doing
  - i.e. if you are going off the end of an array its deliberate and not a mistake

- If you do not know what you are doing C can be dangerous…
  - There is no type safety like Java or Haskell
  - You can do strange maths with pointers

  - Programmers have been trained to ignore warnings…

How do we fix this?

- Short term:
  - Do not teach programmers unsafe practice
  - Listen to your compiler
- Longer term
  - Maybe we should make it harder to do dangerous things?
  - Language standard, compilers, and tools evolve.

Buffer Overflow

- You get told that functions like *gets* or *strcpy* in C are dangerous and should not be used…
- …some OSs will even start outputting warnings to users!?
- …why?

```
[$ cat test.c
#include <stdio.h>
int main(void) { char *str; gets(str); return 0; }
[$ ./test
warning: this program uses gets(), which is unsafe.
```

=> Buffer overflow vulnerability

Buffer Overflow

Several decades old problem (still appears in SANS TOP 25 Software errors!!)

Main cause: putting more data than *intended*!!

Consequences: memory corruption (can be very dangerous!)

Buffer Overflow

- What happens when you declare array?
  - You get a region of memory
- Pointers are used to address arrays
  - Very easy to fall off the end of the region!

- Have been known about since the dawn of computers, but earliest tutorial on how to exploit them in *Phrack magazine*

- *Smashing the Stack for Fun and Profit by Aleph1*

http://phrack.org/issues/49/14.html

Example

```
example1.c:
------------------------------------------------------------------------------
void function(int a, int b, int c) {
   char buffer1[5];
   char buffer2[10];
}

void main() {
  function(1,2,3);
}
------------------------------------------------------------------------------
```

```
bottom of                                                            top of
memory                                                               memory
           buffer2       buffer1   sfp   ret   a     b     c
<------   [            ][        ][    ][    ][    ][    ][    ]

top of                                                            bottom of
stack                                                                 stack
```

example2.c

```
------------------------------------------------------------------------------
void function(char *str) {
   char buffer[16];

   strcpy(buffer,str);
}

void main() {
  char large_string[256];
  int i;

  for( i = 0; i < 255; i++)
    large_string[i] = 'A';

  function(large_string);
}
------------------------------------------------------------------------------
```

```
example2.c
------------------------------------------------------------------------------
void function(char *str) {
   char buffer[16];

   strcpy(buffer,str);
}

void main() {
  char large_string[256];
  int i;

  for( i = 0; i < 255; i++)
    large_string[i] = 'A';

  function(large_string);
}
------------------------------------------------------------------------------
```

```
bottom of                                                              top of
 memory
                  buffer                   ret    *str                 memory
<------           sfp                    ][    ][    ]
                 [AAAAAAAAAAAAAAAA][
top of                                                              bottom of
 stack                                                                  stack
```

```
example2.c
------------------------------------------------------------------------------
void function(char *str) {
   char buffer[16];

   strcpy(buffer,str);
}

void main() {
  char large_string[256];
  int i;

  for( i = 0; i < 255; i++)
    large_string[i] = 'A';

  function(large_string);
}
------------------------------------------------------------------------------
```

```
bottom of                                                            top of
 memory
                  buffer     sfp         ret                         memory
<------          *str  [AAAAAAAAAAAAAAAA][AAAA][
                 ][          ]
top of                                                            bottom of
 stack                                                                stack
```

```
example2.c
------------------------------------------------------------------------------
void function(char *str) {
   char buffer[16];

   strcpy(buffer,str);
}

void main() {
  char large_string[256];
  int i;

  for( i = 0; i < 255; i++)
    large_string[i] = 'A';

  function(large_string);
}
------------------------------------------------------------------------------
```

```
bottom of                                                              top of
 memory
                  buffer            sfp   ret   *str                   memory
<------          [AAAAAAAAAAAAAAAA][AAAA][AAAA][     ]

top of                                                              bottom of
 stack                                                                  stack
```

```
example2.c
------------------------------------------------------------------------------
void function(char *str) {
   char buffer[16];

   strcpy(buffer,str);
}

void main() {
  char large_string[256];
  int i;

  for( i = 0; i < 255; i++)
    large_string[i] = 'A';

  function(large_string);
}
------------------------------------------------------------------------------
```

```
bottom of                                                                   top of
 memory                                                                     memory
                  buffer     sfp          ret          *str  [AAAAAAAAAAAAAAAA][AAAA]
<------          [AAAA][AAAA]AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA

top of                                                                   bottom of
 stack                                                                       stack
```

BoF Consequences

- At this point the program *probably* crashes

- Unless 0xAAAA contains valid program code… the CPU can’t run from there so you’ll *probably* get an illegal instruction exception

What happens ?

- Lets say this overflow happens…
  - …maybe you corrupt some local stack data
  - …maybe you overflow onto some protected memory region and trigger a segfault?
- Suppose you *don't* trigger a segfault…
  - …what happens when the function returns?

```
example2.c
------------------------------------------------------------------------------
void function(char *str) {
   char buffer[16];

   strcpy(buffer,str);
}

void main() {
  char large_string[256];
  int i;

  for( i = 0; i < 255; i++)
    large_string[i] = 'A';

  function(large_string);
}
------------------------------------------------------------------------------
```

```
bottom of                                                                 top of
 memory
                  buffer            sfp   ret   *str                      memory
<------          [AAAAAAAAAAAAAAAA][AAAA][AAAA][    ]
                                               |
top of                                                                 bottom of
 stack                                         V                          stack
                                              rip
```

BoF Consequences

- But we kind of know where some stuff is in memory…
  - …the stack (in particular) is fairly predictable
- …and we control what we put into that buffer…
  - …so we *could* put valid instruction sequences into it
- …in which case we could make the program start to run our own code instead of its own…

BoF Countermeasures

- Modern CPUs don’t allow you to write to regions of memory you can execute, or execute from regions of memory you can write to
- But you can get round this…
  - Return to libc or ROP (We’ll cover them in Software and Systems Security in year 4)
- Stack canaries help prevent exploitation
  - Stick a random number before the return address… check it hasn’t changed before returning
- Shadow stacks also help
  - Keep a second stack with just the return addresses on… check its consistent with the main stack
  - Not implemented everywhere

BoF Countermeasures

- If you’re using C use the bounded strcpy/gets variants, use safe version of C APIs

  strnpcy is better than strcpy

  fgets is better than gets

A quick read:

https://security.web.cern.ch/recommendations/en/codetools/c.shtml

Format String Errors

- Formatted output functions consist of a format string and a variable number of arguments.

- The format string provides a set of instructions that are interpreted by the formatted output function.

By controlling the content of the format string a user can control execution of the formatted output function.

Format String

Format strings

- Format strings are character sequences consisting of *ordinary characters* and *conversion specifications*.
- Conversion specifications convert arguments according to a corresponding conversion specifier, and write the results to the output stream.
- Conversion specifications begin with a percent sign (%) and are interpreted from left to right.
- If there are more arguments than conversion specifications, the extra arguments are ignored.
- If there are not enough arguments for all the conversion specifications, the results are undefined.

Format String

- Example functions

```
vfprintf()      fprintf()
vprintf()       printf()
vsprintf()      sprintf()
vsnprintf()     snprintf()
```

Format String

```
$ cat example3.c && make example3
#include <stdio.h>

int main(int argc, char *argv[]) {
  printf("This program is called: ");
  printf(argv[0]);
  printf("\n");

  return 0;
}
cc  -I/opt/homebrew/opt/openjdk/include   example3.c   -o example3
example3.c:5:10: warning: format string is not a string literal (potentially insecure) [-Wformat-security]
  printf(argv[0]);
         ^~~~~~~
example3.c:5:10: note: treat the string as an argument to avoid this
  printf(argv[0]);
         ^
         "%s",
1 warning generated.
$
```

Format String

man 3 printf

BUGS top

Because **sprintf**() and **vsprintf**() assume an arbitrarily long string, callers must be careful not to overflow the actual space; this is often impossible to assure. Note that the length of the strings produced is locale-dependent and difficult to predict. Use **snprintf**() and **vsnprintf**() instead (or asprintf(3) and vasprintf(3)).

Code such as **printf(***foo***);** often indicates a bug, since *foo* may contain a % character. If *foo* comes from untrusted user input, it may contain **%n**, causing the **printf**() call to write to memory and creating a security hole.

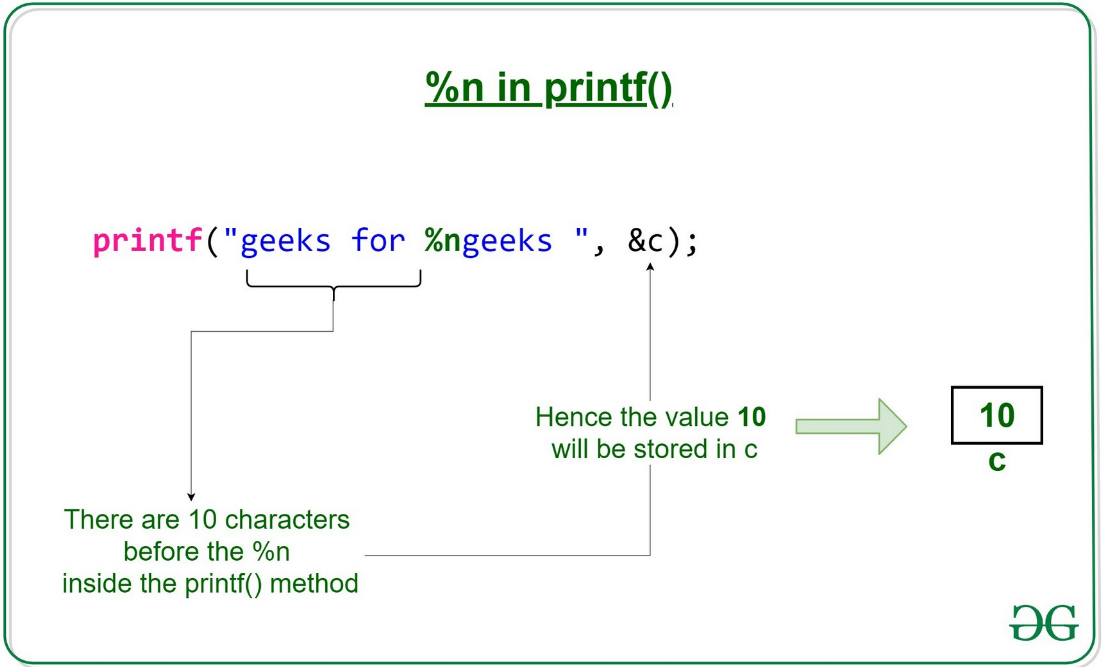
Format String

%n ?

- LibC's *printf* function handles formatted output…
  - %s prints a string…
  - %d or %i prints a decimal integer…

```
n       The number of characters written so far is stored into the
        integer pointed to by the corresponding argument.  That
        argument shall be an int *, or variant whose size matches
        the (optionally) supplied integer length modifier.  No
        argument is converted.  (This specifier is not supported
        by the bionic C library.)  The behavior is undefined if
        the conversion specification includes any flags, a field
        width, or a precision.
```

Format String



https://www.geeksforgeeks.org/g-fact-31/

Format String, How ?

- Say you wanted to do columnar output (and were really weird)
- Say we have an address book we want to print like the following

```
Sana Belguith: 3.16 MVB, Bristol
                01234 567890
                Born 1988-03-15
```

```
#include <stdio.h>

void print_address_book(char *name,
                        char *data[]) {
  int align;
  printf("%s: %n", name, &align);
  if (data != NULL)
    do {
      printf("%s\n", *data);
      for (int i = 0; i < align; i++)
        putchar(' ');
    } while (*(++data) != NULL);
  putchar('\n');
}
```

Format String, How ?

- Say you control the format string…
- Format string arguments typically passed via the stack

- What happens if you print more arguments than you have?

```
#include <stdio.h>
int main(void) {
  int target = 0x31337;
  char *args =
    "01: %p\n02: %p\n03: %p\n04: %p\n"
    "05: %p\n06: %p\n07: %p\n08: %p\n"
    "09: %p\n0a: %p\n0b: %p\n0c: %p\n"
    "0d: %p\n0e: %p\n0f: %p\n10: %p\n";
  printf(args);
  return 0;
}
```

Format String, How ?

```
$ ./example5
01: 0x16f147700
02: 0xd6eb3c
03: 0x100cbbf44
04: 0x31337
05: 0x16f147850
06: 0x100d690f4
07: 0x0
08: 0x0
09: 0x0
0a: 0x0
0b: 0x0
0c: 0x0
0d: 0x100dc8138
0e: 0x0
0f: 0x4d55545a
10: 0x20a000000000
```

```c
#include <stdio.h>
int main(void) {
  int target = 0x31337;
  char *args =
    "01: %p\n02: %p\n03: %p\n04: %p\n"
    "05: %p\n06: %p\n07: %p\n08: %p\n"
    "09: %p\n0a: %p\n0b: %p\n0c: %p\n"
    "0d: %p\n0e: %p\n0f: %p\n10: %p\n";
  printf(args);
  return 0;
}
```

Format String Consequences

- By careful choice of format string we can write to arbitrary addresses somewhere after the stack pointer…
- This could be a local variable…
  - Data corruption
- This could be return address…
  - Control flow corruption and arbitrary code execution

Going further…

- See *Exploiting format string vulnerabilities* by scut/team teso
- See *Exploiting a format string bug in Solaris CDE* (Phrack Magazine, Volume 0x16, Issue 0x46) by Marco Ivaldi

- *(or take Systems and Software Security in Year 4 ;-))*