OntarioTech Business & IT

Low Level Memory Vulnerabilities

By

Dr. Madani

Winter 2024

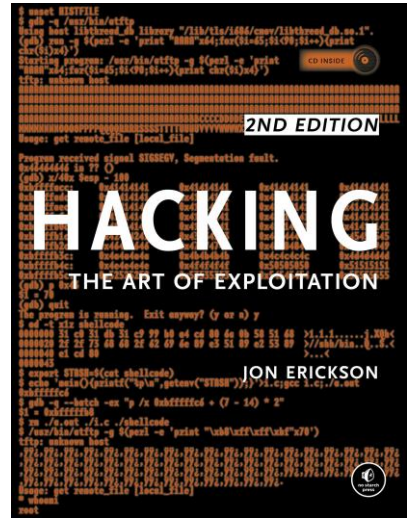
Learning Objectives

Upon completion of this material, you should be able to

- Describe memory layout of a Linux process.
- Describe low-level memory vulnerabilities such as Bufferoverflow and Stackoverflow.
- Describe Fixed-width and Floating-point precision vulnerabilities.
- Protect against low-level memory vulnerabilities.

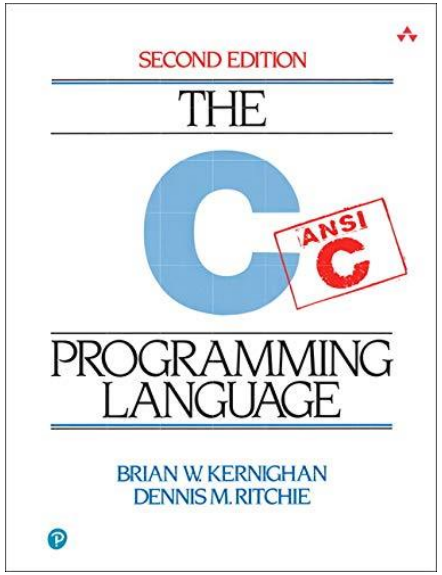
Optional Reading

- Chapters 9 of Designing Secure Software by Kohnfelder
- Chapter 0x300 of Hacking the art of Exploitation by Jon Erickson



The C Programming Language

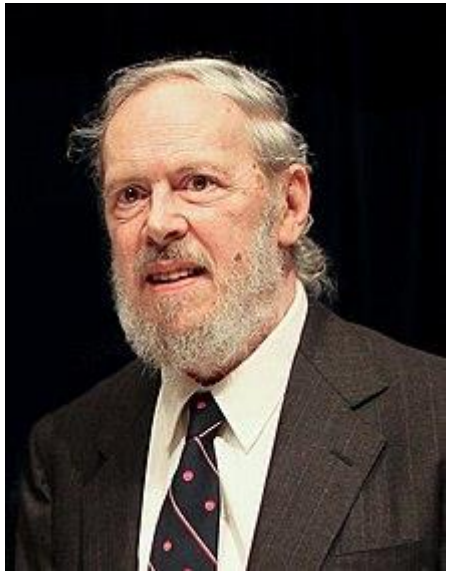
- Note: we are not going to teach the entire “C” programming language. Students are expected to have familiarity with the language.
- We are going to do a quick refresher on “C” …
- We will discuss programs written in “C” and compiled for Linux environment. But, without loss of generality, all the discussed issues will be equally applied to all other “unmanaged memory access” programming languages and other runtime environments.



https://github.com/auspbro/ebook-c/blob/master/The.C.Programming.Language.2Nd.Ed%20Prentice.Hall.Brian.W.Kernighan.and.Dennis.M.Ritchie..pdf

The C Programming Language – cont'd

- The language was created in 1970s by Dennis Ritchie.
  - General purpose programming language, imperative procedural, statically typed.
  - It is a compiled language and provides "low-level" access to memory.



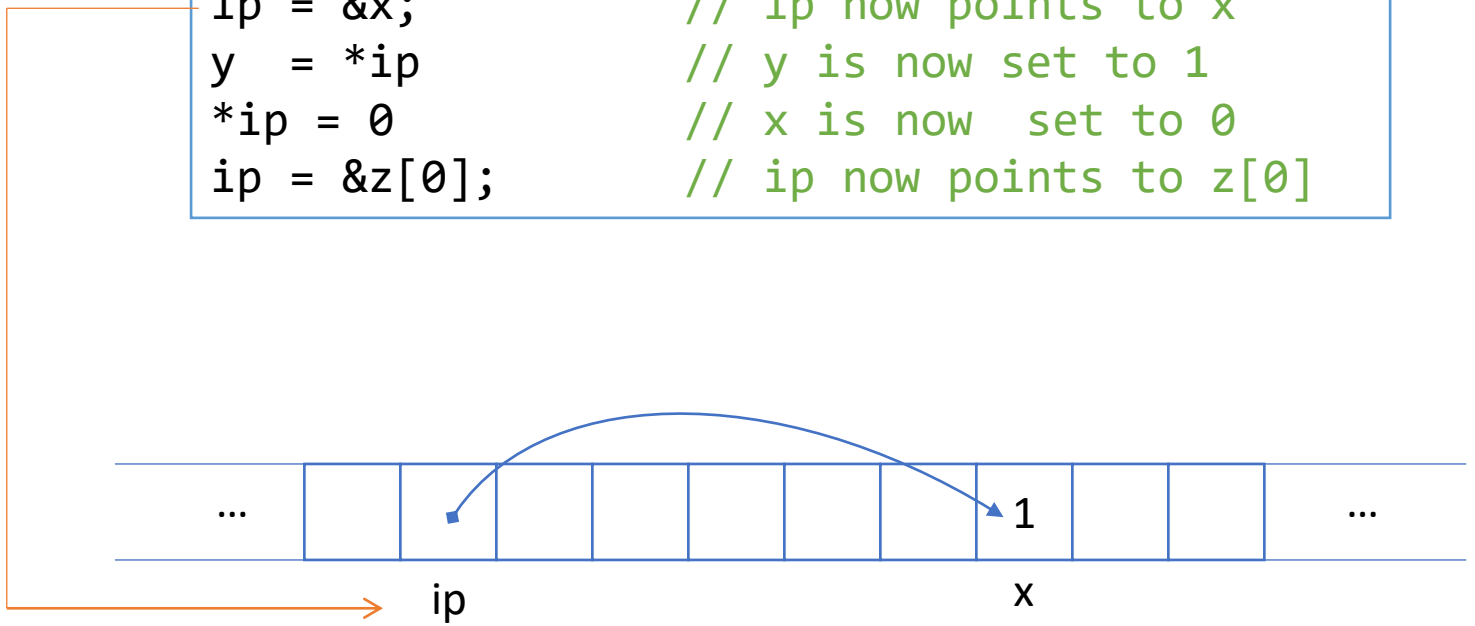
Programming Language	2023	2018	2013	2008	2003	1998	1993	1988
Python	1	5	8	7	13	28	17	-
C	2	2	1	2	2	1	1	1
Java	3	1	2	1	1	17	-	-
C++	4	3	4	3	3	2	2	6
C#	5	4	5	8	12	-	-	-
Visual Basic	6	15	-	-	-	-	-	-
JavaScript	7	7	10	9	8	21	-	-
Assembly language	8	12	-	-	-	-	-	-
SQL	9	-	-	-	7	-	-	-
PHP	10	8	6	5	6	-	-	-
Objective-C	16	18	3	45	47	-	-	-
Ada	29	27	17	18	15	7	8	2
Lisp	31	31	13	15	14	9	5	3
Pascal	242	128	15	20	99	11	3	7
(Visual) Basic	-	-	7	4	4	3	6	5

https://www.tiobe.com/tiobe-index/

Pointers and Their Powers

- Pointer
  - Is a variable that contains the address of a variable.
  - The unary operator “&” ives address of an object.
    - Example:

```
int x = 1, y=2, z[10];
int *ip;           // ip is a pointer to int
ip = &x;           // ip now points to x
y  = *ip           // y is now set to 1
*ip = 0            // x is now  set to 0
ip = &z[0];        // ip now points to z[0]
```



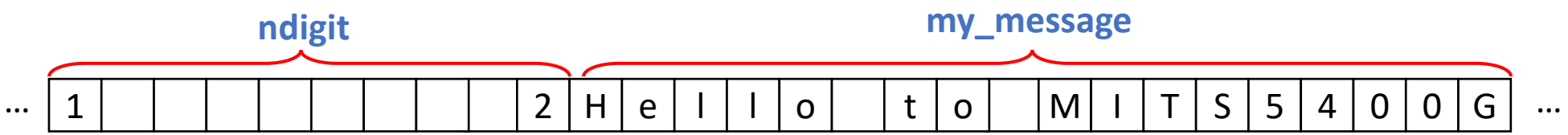
Arrays and Their Problems

- Array
  - Clubbing multiple entities (e.g., variables) of similar types into a larger group.
  - Allows the application of operations to an entire set of values at once!

- Creation, E.g.:
  - `int ndigit[10];  // list of 10 integers`
  - `char my_message[] = "Hello to MITS5400G"`

Assignment, E.g.:

```
ndigit[0] = 1   //assigning value 1 to the first slot
ndigit[9] = 2   //assigning value 2 to the last slot
```



Buffer Overflow

- Buffer
  - Is a contiguous memory associated with a variable or field
  - Common in C
    - All strings are (NUL-terminated) arrays of char's
- Overflow
  - Put more into the buffer than it can hold!
- Where does the overflowing data go?
  - According to C standards, such behaviour is "undefined". Effectively is allowed to do anything!
  - Most compilers assume the program does not have any overflow, and so the program will a access whatever happens to be at the access location!

Buffer Overflow – con't

- Example 1
  - Can you see the problem? We are trying to copy 8 character-long string into a 4-character long buffer! --- Overflow
  - Let's see how is that look like on the stack:

After "strcpy" (string copy function in stdio.h) is executed:


```
void func(char *arg1)
{
    char buffer[4];
    strcpy(buffer, arg1);
    ...
}

int main()
{
    char *mystr = "AuthMe!";
    func(mystr);
    ...
}
```

So when we get at the end of the "func", we are going to do the same process as we always do, to return to the calling function "main". But the "Frame Pointer" is now corrupted and will set it to whatever the strange value is, and we get "SEGMENTATION FAULT" when we use that frame pointer!

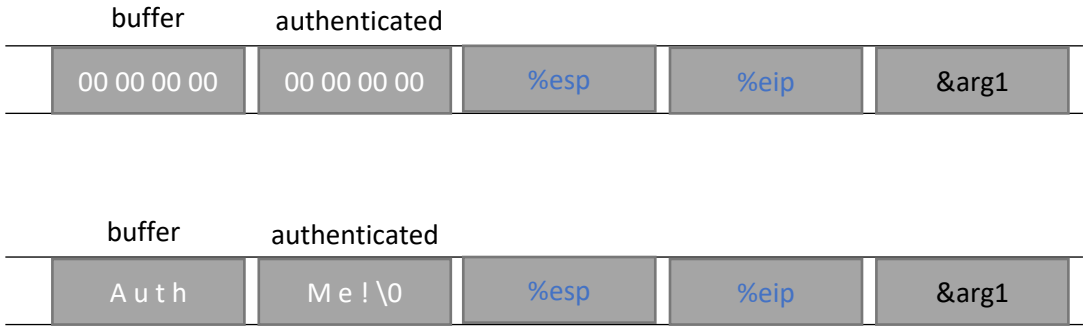
Normally, you would think, oh too bad! That's a crash, this is a bug, we are going to eventually find it and fix it!

Buffer Overflow – con't

```
void func(char *arg1)
{
    int authenticated = 0;
    char buffer[4];
    strcpy(buffer, arg1);
    if(authenticated) { ...
}

int main()
{
    char *mystr = "AuthMe!";
    func(mystr);
    ...
}
```

- Example 2
  - Can you see the security implication in this code snippet when the buffer is overflowed?
  - "func" Stack Frame before execution of strcpy:
  - "func" Stack Frame after execution of strcpy:



  - Now, the content of "authenticated" variable is overwritten, and consequently `if(authenticated)` will be true! Result in program doing things that we probably didn't intent

Buffer Overflow – con't

- The discussed examples provide their own strings. However, in reality strings come from users in myriad aways:
  - Text input
  - Packets
  - Environment Variables
  - File Input …
- Validating assumptions about user input, therefore, is extremely important.

Code Injection

- The main idea – using buffer overflow for code injection:

  Challenge 1

  Load “our malicious” code into memory (sometimes referred to as payload).

  - Important to note, that this code must be machine code instructions (i.e., already compiled and ready to run on the victim’s machine!)
  - We have to be careful in how we construct it:
    - It can’t contain any all-zero bytes, otherwise sprint/gets/scanf/ … will stop copying.
      - But, how can you write assembly code to never contain any full zero byte?
    - It can’t use the loader (we’re injecting)

  Challenge 2

  Somehow get %eip to point to it.

News Flash!

Both challenges are easily solvable

Integer Overflow

- In C, a variable has a maximum value, and when that value is exceeded, the variable's value will wraparound.

```
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);
  }
```

In this example, we are reading a number from the network.
And suppose the adversary send a very large number, e.g., 1,073,741,824
Note: sizeof(char *) is 4 bytes!

Then, it means the adversary has arranged that `nresp*sizeof(char *)` overflows to become "0".
So "malloc" will allocate size-0 buffer!

So, the subsequent writes to allocated response overflow it!
Of curse, this overflow can be controlled by the adversary and potentially inject some code, or modify existing variable contents allocated on the heap and/or stack!

Read Overflow

- So far, we have looked into “writing past the end of a buffer” – what if a bug could permit “reading past the end of a buffer?”
  - Might leak secret information.
  - Example: this program will “read-in” a message into “buf” and then will echo back what was read.

(1) We get the user to tell what’s the length of the message and storing it as an integer value.

(2) then, we read in the message into buf

(3) finally, we echo back the message up to the length specified by the user.

```
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");
  }}
```

Can you see the problem? The specified length may exceed actual message length!

Read Overflow – cont'd

- Heartbleed Bug is an example of read overflow that we just discussed!

  - https://en.wikipedia.org/wiki/Heartbleed
  - By some estimated, effected nearly 600k servers over the Internet.
  - The SSL (i.e., TLS) protocol allows the clients to send "heartbeat" messages to the server and asking it to echo back to confirm the connection is still active.
  - The heartbeat message specifies the length of its echo-back portion, but buggy SSL software did not check the length was accurate!
    - In fact, it could be much larger than the echo back message.
    - Thus, an attacker could request a longer length and read past the contents of the message buffer!
      - Leaking passwords, crypto keys, … - DATA IN USE!

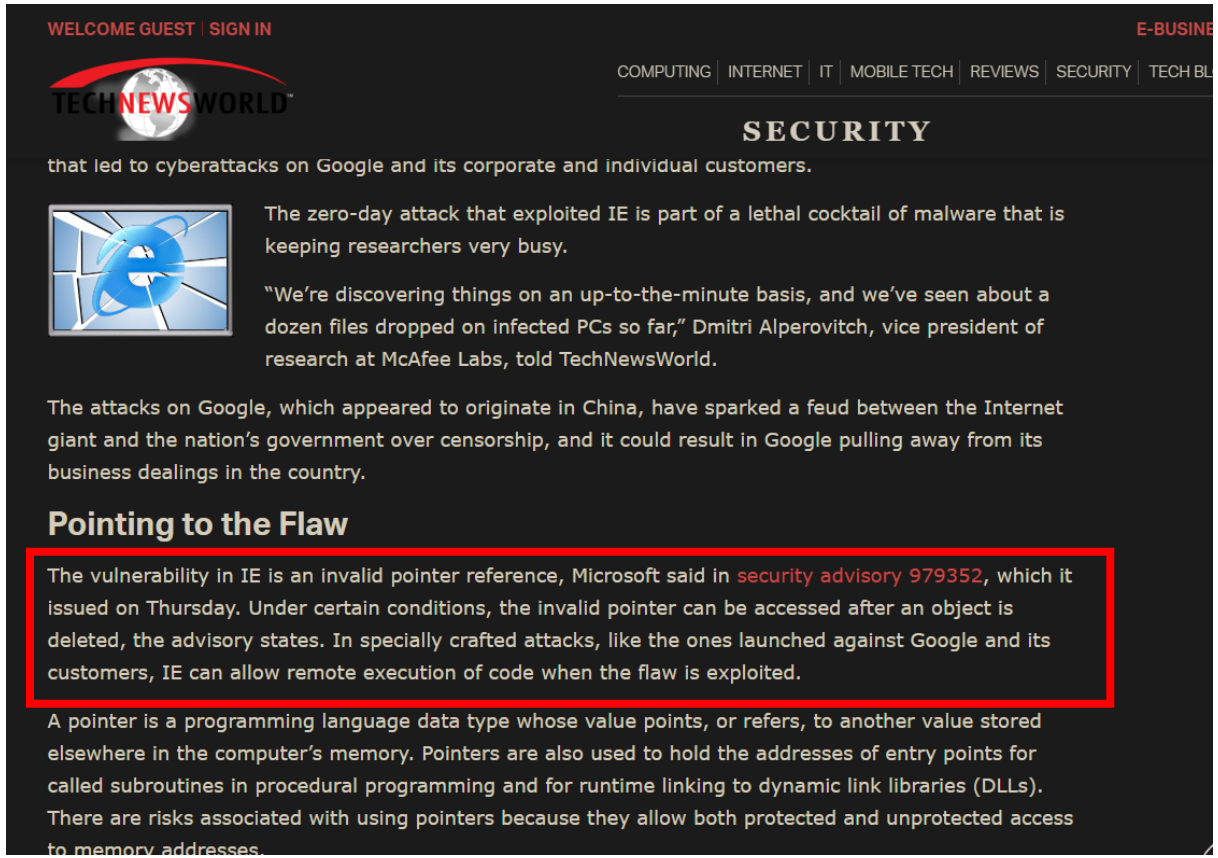
Stale Memory

- 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 dereferences, it will access attacker-controlled data.

```
struct foo { int (*cmp)(char*, char*); }
struct foo *p = malloc(…);
free(p);
…
q = malloc(…); //reuses memory that was given to p
*q = 0xdeadbeef; //attacker may control this value.
…
p->cmp(“hello”,”goodby”); //dangling ptr mistakenly used by the program later on!
```

Stale Memory – cont'd

- In fact it was this sort of bug, that played a huge role in the attack that China executed against Google back in 2010 ! https://www.technewsworld.com/story/ies-role-in-the-google-china-war-69121.html



WELCOME GUEST | SIGN IN

E-BUSINE

TECHNEWSWORLD

COMPUTING | INTERNET | IT | MOBILE TECH | REVIEWS | SECURITY | TECH BL

SECURITY

that led to cyberattacks on Google and its corporate and individual customers.

The zero-day attack that exploited IE is part of a lethal cocktail of malware that is keeping researchers 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 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.

A pointer is a programming language data type whose value points, or refers, to another value stored elsewhere in the computer's memory. Pointers are also used to hold the addresses of entry points for called subroutines in procedural programming and for runtime linking to dynamic link libraries (DLLs). There are risks associated with using pointers because they allow both protected and unprotected access to memory addresses.

Common elements

- The discussed attacks have some common elements:
  1. The attacker is able to control some data that is used by the program.
     - It come as a form of a web request to a server or a user input from a terminal!
  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

- The discussed attacks violate memory and type safety.

Memory Safety

- A memory safe program execution:
  1. Temporal Safety: only creates pointers through standard means
     - `p=malloc(…), or p = &x, or p= &buf[4], etc.`
  2. Spatial Safety: only uses pointer to access memory that "belongs" to that pointer.

- Memory Safe Program
  - is a program that all of its possible executions (i.e., executions over all possible inputs) are memory safe.
- Memory Safe Programming Language
  - a language is memory safe if all possible programs that can be written in that language are memory safe.

Memory Safety

- Most modern languages (except for C/C++) such as C#, Java, GoLang, Python are memory safe !
  - Get a good security boost by using memory safe programming languages.
  - But C/C++ are here to stay! Tons of legacy systems are out there!
- Can you add memory safety checks to C/C++ programs at compile time? Yes, however, the runtime performance overheads has been the limiting factor
  - Read more on CCured https://web.eecs.umich.edu/~weimerw/p/p232-condit.pdf
  - Softbound/CETS https://people.cs.rutgers.edu/~sn349/softbound/

Type Safety

- Each object is ascribed a type (int, pointer to int, …)
- Operations on the objects are always compatible with the object's type.
  - Example of an undefined behavior is when you over run a buffer in C/C++.
- Dynamically Typed Languages
  - Like Ruby and Python, which do not require declarations that identify types can be viewed as type safe as well.
  - Each object has one type: Dynamic
    - Each operation on a Dynamic object is permitted but may be unimplemented. In this case, it throws an exception.

Make the Bug Harder to Exploit

- Recall the steps of a stack smashing attack:
  1. Putting attacker code into the memory (no zeroes)
     - Defense: Make this detectable with stack canaries!
  2. Getting %eip to point and run the attacker code
     - Defense: Make stack (and heap) non-executable! (so even if canaries could be bypassed, no code loaded by the attacker can be executed!) [unfortunately this approach can be bypassed too, using return-to-lib-c]
       - Defense against return-to-libc: User address-space layout randomization. Randomly place standard libraries and other elements in memory, making them harder to guess!
- How can we make these attack steps more difficult?

Address Space Layout Randomization (ASLR)

- Is a technique involved in preventing exploitation of memory corruption vulnerabilities.
- It randomly arranges the address space positions of key data areas of a process (e.g., location of standard library functions) in order to prevent an attacker from reliably jumping to a particular exploited function in memory.
  - First introduced on Linux 2004 and adoption on other systems came slowly afterwards; most by 2011.

- ASLR is not perfect
  - only shifts the offset of memory areas not location within those areas.
  - May not apply to program code, just libraries.
  - Need sufficient randomness, or can brute force
    - 32-bit systems typically offer 16 bits possible starting positions. Shacham demonstrated a bruteforce attack could defeat such randomness in 216 seconds (on 2004 hardware) https://hovav.net/ucsd/dist/asrandom.pdf **[good project topic]**
    - 64-bit systems more promising, e.g., 40 bits possibility.

Return Oriented Programming (ROP)

- Outside of scope, but makes it a “great” project topic: https://hovav.net/ucsd/talks/blackhat08.html#:~:text=We%20describe%20return%2Doriented%20programming,with%20a%20%E2%80%9Cret%E2%80%9D%20instruction.
  - Uses machine codes existing in the program code, to build a custom malicious function
  - You can construct “Turing complete” programs!
  - Another great paper for attacking 64 bit machines using ROP https://ieeexplore.ieee.org/document/6956567 [consider this another great project topic)
  - It could defeat stack canneries and randomization – an actual exploit was developed for nginx in 2014!
- “Control Flow Graph” (CFG) also makes a great project topic: https://research.cs.cornell.edu/crystal/fastcheck/fastcheck-pldi07.pdf

Secure Coding Practices

- While automated defenses are ideal, but one cannot solely rely on them to prevent exploitation of vulnerabilities.
- Thus, developers must use discipline, limiting themselves to coding patterns that avoid discussed vulnerabilities.
  - Such patterns are documented in coding standards such as https://wiki.sei.cmu.edu/confluence/display/c
- Combined with advanced code review and testing (will discuss these later in the course!)

Secure Coding Practices

- Rule 1 - Enforce input compliance
  - Do not trust any user input, always check and validate.
  - Check for out-of-band access to arrays.
- Rule 2 – Use safe string functions
  - Traditional string library routines assume target buffers have sufficient length.

```
char str[4];
char buf[10] = "fine";
strcpy(str,"hello"); // overflows str
strcat(buf,"day to you"); // overflows buf
```

  - Safe versions check the destination length

```
char str[4];
char buf[10] = "fine";
strlcpy(str,"hello",sizeof(str)); //fails
strlcat(buf,"day to you",sizeof(buf));//fails
```

Secure Coding Practices – cont'd

- … for string-oriented functions

```
strcat ⇒ strlcat
strcpy ⇒ strlcpy
strncat ⇒ strlcat
strncpy ⇒ strlcpy
sprintf ⇒ snprintf
vsprintf ⇒ vsnprintf
gets ⇒ fgets
```

- Rule 3 – Don't forget NUL terminator
  - Strings require one additional character to store the NUL terminator. Forgetting that could lead to overflows.

```
char str[3];
strcpy(str,"bye"); // write overflow
int x = strlen(str); // read overflow
```

  - Using safe string library calls will catch this mistake

```
char str[3];
strlcpy(str,"bye",3); // blocked
int x = strlen(str); // returns 2
```

Secure Coding Practices – cont'd

- Rule 4 – Avoid Dangling Pointers by NUL them out!
- Rule 5 – Favor safe libraries
  - E.g., networking: Google protocol buffers, Apache Thrift
  - E.g., vsftp String library