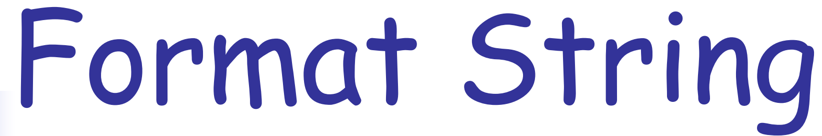
Defending against code injection

Examples

Input Validation

format strings?




- What is a format string vulnerability
- Fundamental "C" problem
- Survey of unsafe functions and how to format strings safely with regular C functions
- Case study: cfingerd 1.4.3 vulnerabilities
- Preventing format string vulnerabilities without programming
- Tools to find string format issues

# What is a Format String?

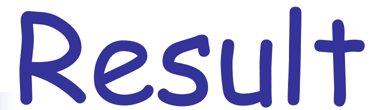
In "C", you can print using a format string:

- printf(const char *format, ...);
- printf("Mary has %d cats", cats);
  - %d specifies a decimal number (from an int)
  - %s would specify a string argument,
  - %x would specify an unsigned uppercase hexadecimal (from an int)
  - %f expects a double and converts it into decimal notation, rounding as specified by a precision argument
  - ...

# Fundamental "C" Problem

- No way to count arguments passed to a "C" function, so missing arguments are not detected
- Format string is interpreted: it mixes code and data
- What happens if the following code is run?

```
int main () {
      printf("Mary has %d cats");
  }
```



# Result

% ./a.out
Mary has -1073742416 cats

- Program reads missing arguments off the stack!
  - And gets garbage (or interesting stuff if you want to probe the stack)

# User-specified Format String

- What happens if the following code is run, assuming there is an argument input by a user?

```
int main(int argc, char *argv[])
  {
     printf(argv[1]);
     exit(0);
  }
```

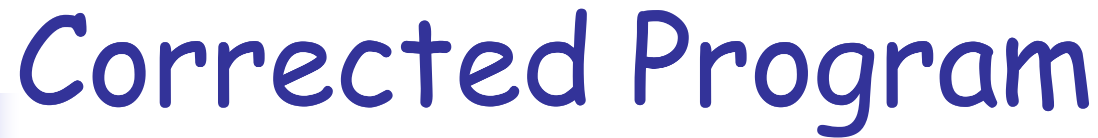
- Try it and input "%s%s%s%s%s%s%s%s%s" How many "%s" arguments do you need to crash it?

# Result

```
% ./a.out "%s%s%s%s%s%s%s"
  Bus error
```

- Program was terminated by OS
  - Segmentation fault, bus error, etc... because the program attempted to read where it was not supposed to
- User input is interpreted as string format (e.g., %s, %d, etc...)
- Anything can happen, depending on input!
- How would you correct the program?

# Corrected Program



```
int main(int argc, char *argv[])
  {
      printf(“%s”, argv[1]);
      exit(0);
  }
```

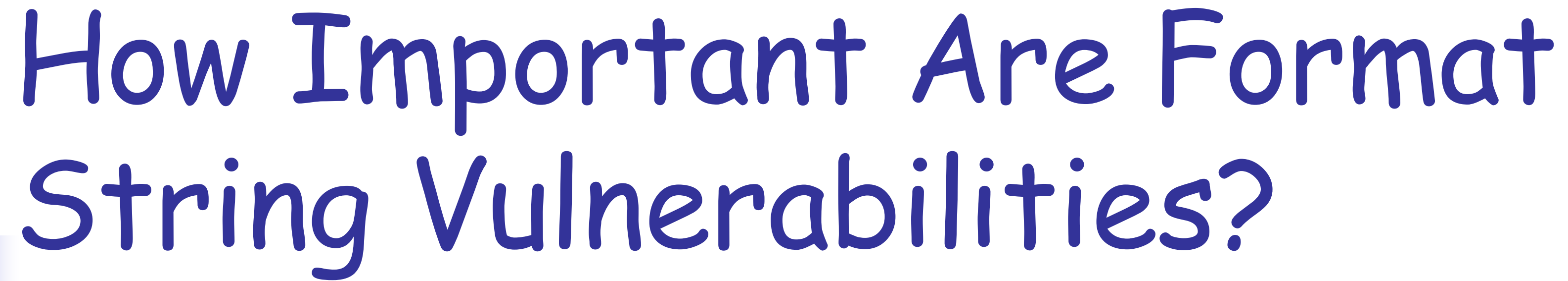
```
% ./a.out "%s%s%s%s%s%s%s"
  %s%s%s%s%s%s%s
```

# Format String Vulnerabilities

- Discovered relatively recently (~2000)
- Limitation of “C” family languages
- Versatile
  - Can affect various memory locations
  - Can be used to create buffer overflows
  - Can be used to read the stack
- Not straightforward to exploit, but examples of root compromise scripts available on the web
  - "Modify and hack from example"

# Definition of a Format String Vulnerability

- A call to a function with a format string argument, where the format string is either:
  - Possibly under the control of an attacker
  - Not followed by appropriate number of arguments
- Difficult to establish whether a data string could possibly be affected by an attacker; considered very bad practice to place a string to print as the format string argument.
  - Sometimes the bad practice is confused with the actual presence of a format string vulnerability



# How Important Are Format String Vulnerabilities?

- Search **National Vulnerability Database** (NIST) for “format string”:
  - Over 890 records overall
  - 107 last 3 years (as of Aug 2020)
- Search **Database** a Mitre (cve.mitre.org) for “format string”:
  - 667 (11 from 2018, 15 from 2017) records of vulnerabilities
- Various applications
  - Databases (Oracle)
  - Unix services (syslog, ftp,...)
  - Linux “super” (for managing setuid functions)
  - cfingerd CAN 2001-0609
- Arbitrary code execution is a frequent consequence

# Functions Using Format Strings

- printf - prints to "stdout" stream
- fprintf - prints to stream
- warn - standard error output
- err - standard error output
- setproctitle - sets the invoking process's title
- sprintf(char *str, const char *format, ...);
  - sprintf prints to a buffer
  - What's the problem with that?

# Better functions than sprintf

Note: do not prevent format string vulnerabilities:

- `snprintf(char *str, size_t size, const char *format, ...);`
  - `sprintf` with length check for "size"
  - Does not guarantee NUL-termination of s on some platforms (Microsoft, Sun)
  - MacOS X: NUL-termination guaranteed
  - Check with "`man sprintf`"

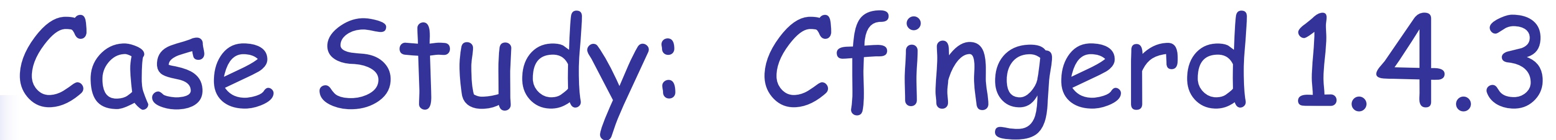
- `asprintf(char **ret, const char *format, ...);`
  - sets `*ret` to be a pointer to a buffer sufficiently large to hold the formatted string .

# Custom Functions Using Format Strings

- It is possible to define custom functions taking arguments similar to printf.
- wu-ftpd 2.6.1 proto.h
  - void reply(int, char *fmt,...);
  - void lreply(int, char *fmt,...);
  - etc...
- Can produce the same kinds of vulnerabilities if an attacker can control the format string

"%n" format command

- Writes a number to the location specified by argument on the stack
  - Argument treated as int pointer
    - Often either the buffer being written to, or the raw input, are somewhere on the stack
      - Attacker controls the pointer value!
  - Writes the number of characters written so far
    - Keeps counting even if buffer size limit was reached!
    - “Count these characters %n”
- All the gory details you don't really need to know:
  - Newsham T (2000) "Format String Attacks"



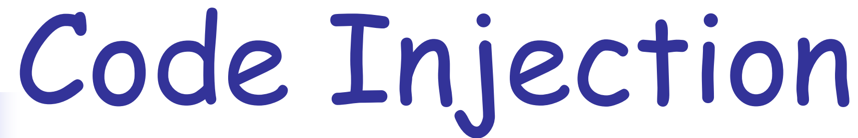
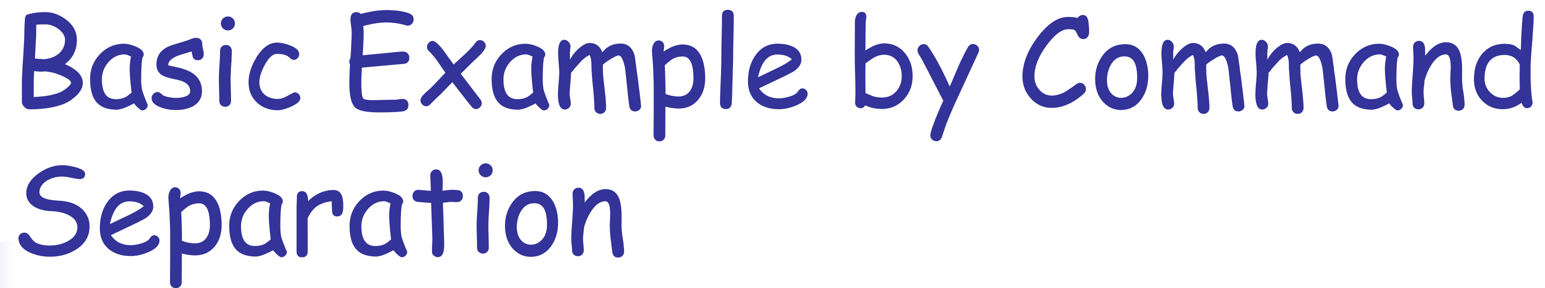
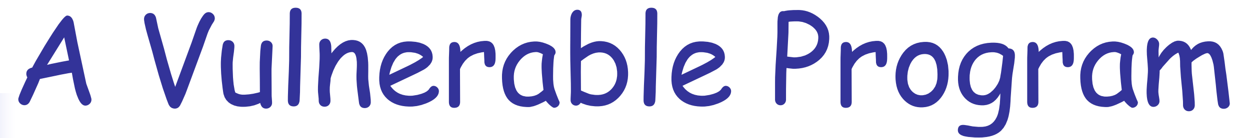
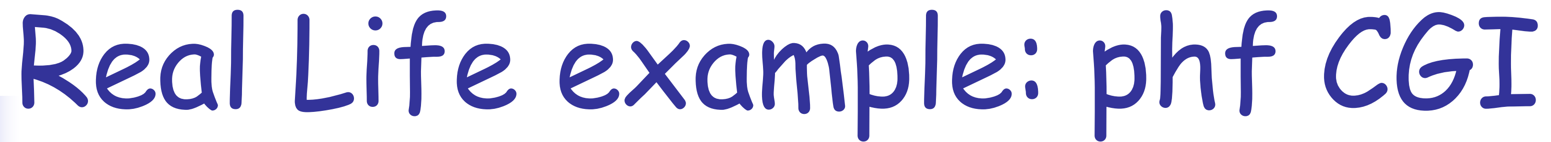
# Case Study: Cfingerd 1.4.3

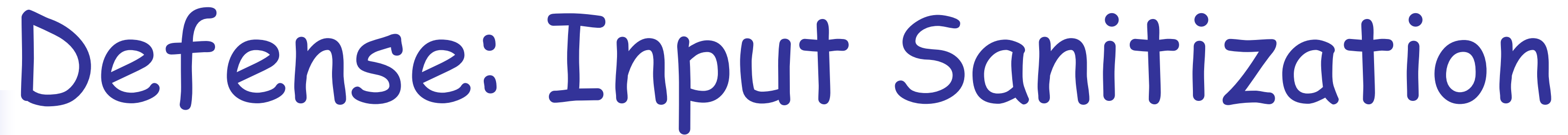
- Finger replacement
  - Runs as root
  - Pscan output: (CAN 2001-0609)
    - defines.h:22 SECURITY: printf call should have "%s" as argument 0
    - main.c:245 SECURITY: syslog call should have "%s" as argument 1
    - main.c:258 SECURITY: syslog call should have "%s" as argument 1
    - standard.c:765 SECURITY: printf call should have "%s" as argument 0
    - etc... (10 instances total)

# Cfingerd Analysis

- Most of these issues are not exploitable, but one is, indirectly at that...
- Algorithm (simplified):
  - Receive an incoming connection
    - get the fingered username
  - Perform an ident check (RFC 1413) to learn and log the identity of the remote user
  - Copy the remote username into a buffer
  - Copy that again into "username@remote_address"
    - remote_address would identify attack source
  - Answer the finger request
  - Log it

# Cfingerd Vulnerabilities

- A string format vulnerability giving root access:
  - Remote data (`ident_user`) is used to construct the format string:
  - ```
    snprintf(syslog_str, sizeof(syslog_str),
        "%s fingered from %s",username, ident_user );
    syslog(LOG_NOTICE, (char *) syslog_str);
    ```
- An off-by-one string manipulation (buffer overflow) vulnerability that
  - prevents remote_address from being logged (useful if attack is unsuccessful, or just to be anonymous)
  - Allows `ident_user` to be larger (and contain shell code)
    ```

# Cfingerd Buffer Overflow Vulnerability

```
memset(uname, 0, sizeof(uname));
  for (xp=uname;
      *cp!='\0' && *cp!='\r' && *cp!='\n'
      && strlen(uname) < sizeof(uname);
      cp++ )
     *(xp++) = *cp;
```

- Off-by-one string handling error
  - uname is not NUL-terminated!
  - because strlen doesn't count the NUL
- It will stop copying when strlen goes reading off outside the buffer

# Direct Effect of Off-by-one Error

```
char buf[BUFLEN], uname[64];
```

- "`uname`" and "`buf`" are "joined" as one string!
- So, even if only 64 characters from the input are copied into "`uname`", string manipulation functions will work with "`uname+buf`" as a single entity
- "`buf`" was used to read the response from the ident server so it *is* the raw input

# Consequences of Off-by-one Error

1) Remote address is not logged due to size restriction:
   - snprintf(bleah, BUFLEN, "%s@%s", uname, remote_addr);
   - Can keep trying various technical adjustments (alignments, etc...) until the attack works, anonymously
2) Not enough space for format strings, alignment characters and shell code in buf (~60 bytes for shell code):
   - Rooted (root compromise) when syslog call is made
     - i.e., cracker gains root privileges on the computer (equivalent to LocalSystem account)

# Preventing Format String Vulnerabilities

1) Always specify a format string
   - Most format string vulnerabilities are solved by specifying "%s" as format string and not using the data string as format string
2) If possible, make the format string a constant
   - Extract all the variable parts as other arguments to the call
   - Difficult to do with some internationalization libraries
3) If the above two practices are not possible, use run-time defenses such as FormatGuard
   - Rare at design time
   - Perhaps a way to keep using a legacy application and keep costs down
   - Increase trust that a third-party application will be safe

# Code Scanners

- Pscan searches for format string functions called with the data string as format string
  - Can also look for custom functions
    - Needs a helper file that can be generated automatically
      - Pscan helper file generator at http://www.cerias.purdue.edu/homes/pmeunier/dir_pscan.html
  - Few false positives



# Code Injection

- Goal: trick program into executing an attacker's code by clever input construction that mixes code and data
- Mixed code and data channels have special characters that trigger a context change between data and code interpretation
  - The attacker wants to inject these meta-characters through some clever encoding or manipulation, so supplied data is interpreted as code

# Code Injection cont.

- Defend against it by using input cleansing and validation; type casts may help if they are possible
- Need to keep track of which data has been cleansed, or keep track of all sources of inputs and cleanse as the input is received

# How widespread? (2020 data)

- Search **National Vulnerability Database** (NIST) for "code injection":
  - Over 900 records overall
  - 360+ last 3 years (recent one Aug 2020)
- Search **Database** a Mitre (cve.mitre.org) for "code injection":
  - 760 (80+ from 2018) records of vulnerabilities



# Basic Example by Command Separation

- ```
  cat >example
  #!/bin/sh
  A = $1
  eval "ls $A"
  ```
- Permissions of file "confidential" before exploit:
  - ```
    % ls -l confidential
    -rwxr-x---  1  user  user  confidential
    ```
- Allow execution of "example":
  - `% chmod a+rx example`
- Exploit (what happens?)
  - `%./example ".;chmod o+r *"`

# Results

- Inside the program, the eval statement becomes equivalent to:
- `eval "ls .;chmod o+r *"`
- Permissions for file "confidential" after exploit:
  - `% ls -l confidential`
    `-rwxr-xr-- 1 user user confidential`
- Any statement after the ";" would also get executed, because ";" is a command separator.
- The data argument for "ls" has become code!

# Other Code Injection by Command Substitution

- (in PHP) Backtick ` `: execution in a command line by command substitution
- `command` gets executed before the rest of the command line
- Imagine a malicious script called "script1":
  - `mkdir oups`
  - `echo oups`
  - `etc...`
- Imagine a program that calls a shell to run grep.
- What happens when this is run?
  - `eval "grep `./script1` afile"`

# Answer

- Script1 is executed
  - first an “oups” directory is created
- The rest of the intended command, “grep oups afile”, is executed



# A Vulnerable Program

```
int main(int argc, char *argv[], char **envp)
   {
    char buf [100];
    buf[0] = '\0';
    snprintf(buf,sizeof(buf),"grep %s text",argv[1]);
    system(buf);
    exit(0);
   }
```

What happens when we run the following?

```
% ./a.out \`./script\`
```

# Answer

- The program calls
  - `system("grep `./script` text");`
  - can be verified by adding "`printf( "%s", buf)`" to the program
- So we could make a.out execute any program we want
  - Imagine that we provide the argument remotely
  - Anyone running a.out would run arbitrary code as the owner of a.out
  - What if a.out runs with root privileges?

# Shell Metacharacters

- '`' to execute something (command substitution)
- ';' is a command ("pipeline") separator
- '&' start process in the background
- '|' is a pipe (connecting standard output to standard input)
- '&&' , '||' logical operators AND and OR
- '<<' or '>>' prepend, append
- # to comment out something

Refer to the appropriate man page (man csh) for all characters

- How else can code be injected into a.out?

# Defending Against Code Injection

- Input cleansing and validation
  - Model the expected input
    - Discard what does not fit (e.g., metacharacters)
  - Keep track of which data has been cleansed
    - e.g., Perl's taint mode
  - Keep track of all sources of inputs
    - Or cleanse as the input is received
- Type and range verification, type casts
- Separating code from data
  - Transmit, receive and manipulate data using different channels than for code

# Input Cleansing

- Key to preventing code injection attacks
- Common problem where code is generated dynamically from some data
  - SQL (database Simple Query Language)
  - System calls and equivalents in PHP, Windows CreateProcess, etc...
  - HTML may contain JavaScript (Cross-site scripting vulnerabilities)

# Intuitive Approach

Block or escape all metacharacters

- but what are they?

Problems:

- Character encodings
  - octal, hexadecimal, UTF-8, UTF-16, binary, Base-64, URL encoding, ...
- Obfuscation
  - Escaped characters that can get interpreted later
  - Engineered strings such that by blocking a character, something else is generated

# Wrong Way to Cleanse Input (Sanitize)

```
int main(int argc, char *argv[], char **envp) {
   static char bad_chars[] = "/ ;[]<>&\t";
   char *user_data;
   char *cp;
   /* Get the data */
   user_data = getenv("QUERY_STRING");
   /* Remove bad characters. WRONG! */
   for (cp = user_data; *(cp += strcspn(cp,bad_chars));
   /* */)
    *cp = '_';
…
```

- http://www.cert.org/tech_tips/cgi_metacharacters.html



# Real Life example: phf CGI

CVE-1999-0067

```
strcpy(commandstr, "/usr/local/bin/ph -m ");
  escape_shell_cmd(serverstr);
  strcat(commandstr, serverstr);
  (...)
  phfp = popen(commandstr,"r");
```

- What could be the problem?
  - besides the potential buffer overflows

# Real Life example: phf CGI cont. Black List of Characters

```
void escape_shell_cmd(char *cmd) {
      (...)
      if(ind("&;`'\"|*?~<>^()[]{}$\\"
      ,cmd[x]) != -1){
      (...)
  }
```

- Author forgot to list newlines in "if" statement...
- Exploit: input “newline” and the commands you want executed...

# More Robust Cleansing

- ```
  {...}
  static char ok_chars[] =
          "1234567890!@%-_=+:,./\
           abcdefghijklmnopqrstuvwxyz\
           ABCDEFGHIJKLMNOPQRSTUVWXYZ";
                  {...}
  for (cp = user_data; *(cp += strspn(cp, ok_chars)); /* */ )
                  *cp = '_';
  ```

- http://www.cert.org/tech_tips/cgi_metacharacters.html
- a.k.a. White List vs Black List design principle
  ```



# Defense: Input Sanitization

- Do not attempt to list all forbidden characters
  - It is easy to forget and and one missed character leads to defeat
- Make a list of all allowed characters
  - Without metacharacters
- Convert to a variable of numerical type, if a number is expected
- Truncate input strings if the expected length is known

# Other Input Validation Issues

- Range of types
  - Short vs long integers
  - Unsigned vs signed
- Integer overflows
  - Validate range (e.g., array indexes)
    - Attacks can make something negative to reach forbidden data
    - Attacks can reset a counter to zero
      - Data structure reference count vs garbage collection
- Strings in numerical inputs
  - e.g., PHP will accept both string and numerical values for a variable, which may allow unexpected attacks
  - Use typecasts

# Order for Cleansing and Input Validation

1) Resolve all character encoding issues first
2) Cleanse
   - If combinations of characters can produce metacharacters, you may need to do several passes. Example:
     - "a" and "b" are legal if separated from each other, but "ab" is considered a metacharacter. The character "d" is not allowed. After you filter out "d" from "adb", you may be allowing "ab" through the filter!
3) Validate type, range, and format
4) Validate semantics (i.e., meaning of input)