Security

CSC 230 : C and Software Tools

NC State Department of Computer Science

Topics for Today

- Bad programs
  - Path manipulation
  - Log forging
  - Buffer overflow, exploited
- Encryption
- Sources of randomness

Island of Bad Programs

- C gives us a lot of power to write high-performance code
  - It's like programming without training wheels
- But, with great power …
- It's easy to write programs with leaks, buffer overflows and security vulnerabilities
- I remember something about what causes this:
  - Programming errors
  - Failure to validate inputs
  - Inadequate protection for secret information
  - Poor understanding of the operating environment

Island of Bad Programs

```
/** Read an entire block of data from fp.
    Return pointer to the block on success,
    or NULL on failure */
char *getBlock( FILE *fp ) {
   char *buf = (char *)malloc( BLOCK_SZ );
   if (!buf)
      return NULL;
   if (fread( buf, 1, BLOCK_SZ, fp ) != BLOCK_SZ )
      return NULL;
   else
      return buf;
}
```

Island of Bad Programs

```
/** Read an entire block of data from fp.
    Return pointer to the block on success,
    or NULL on failure */
char *getBlock( FILE *fp ) {
   char *buf = (char *)malloc( BLOCK_SZ );
   if (!buf)
      return NULL;
   if (fread( buf, 1, BLOCK_SZ, fp ) != BLOCK_SZ )
      return NULL;
   else
      return buf;
}
```

I think you might have a memory leak here.

- Be sure to clean up resources on all paths out of a function

Island of Bad Programs

```
/** Read the contents of the named text file into
    the given buffer.  If successful, return true.*/
bool getFile( char *name, int cap, char *buffer ) {
  FILE *fp = fopen( name, “r” );
  if ( !fp )
    return false;
  int len = fread( buffer, 1, cap, fp );
  return len > 0;
}
```

Island of Bad Programs

```
/** Read the contents of the named text file into
    the given buffer.  If successful, return true.*/
bool getFile( char *name, int cap, char *buffer ) {
  FILE *fp = fopen( name, "r" );
  if ( !fp )
    return false;
  int len = fread( buffer, 1, cap, fp );
  return len > 0;
}
```

You got yourself a file leak right here.

Island of Bad Programs

```
/** Read the contents of the named text file into
    the given buffer.  If successful, return true.*/
bool getFile( char *name, int cap, char *buffer ) {
  FILE *fp = fopen( name, "r" );
  if ( !fp )
    return false;
  int len = fread( buffer, 1, cap, fp );
  fclose( fp );
  return len > 0;
}
```

Easy to fix.

Island of Bad Programs

```
/** Read the contents of the named text file into
    the given buffer.  If successful, return true.*/
bool getFile( char *name, int cap, char *buffer ) {
  FILE *fp = fopen( name, "r" );
  if ( !fp )
    return false;
  int len = fread( buffer, 1, cap, fp );
  fclose( fp );
  return len > 0;
}
```

But, how's the caller supposed to be able to use this buffer. If it's supposed to be a string, it's not null terminated or anything like that.

Island of Bad Programs

```
/** Read a sequence of values from the user,
    returning the sequence and setting len on
    success, and returning NULL on failure. */
int *getValues( int *len ) {
  scanf( "%d", len );
  int *seq = (int *)malloc( *len * sizeof( int ) );
  for ( int i = 0; i < *len; i++ )
    if ( scanf( "%d", seq + i ) != 1 )
      return NULL;
  return seq;
}
```

Island of Bad Programs

```
/** Read a sequence of values from the user,
    returning the sequence and setting len on
    success, and returning NULL on failure. */
int *getValues( int *len ) {
  scanf( "%d", len );
  int *seq = (int *)malloc( *len * sizeof( int ) );
  for ( int i = 0; i < *len; i++ )
    if ( scanf( "%d", seq + i ) != 1 )
      return NULL;
  return seq;
}
```

There's a potential memory leak.

Island of Bad Programs

```
/** Read a sequence of values from the user,
    returning the sequence and setting len on
    success, and returning NULL on failure. /
int *getValues( int *len ) {
  scanf( "%d", len );
  int *seq = (int *)malloc( *len * sizeof( int ) );
  for ( int i = 0; i < *len; i++ )
    if ( scanf( "%d", seq + i ) != 1 ) {
      free( seq );
      return NULL;
    }
  return seq;
}
```

OK. Fixed.

Island of Bad Programs

```
/** Read a sequence of values from the user,
    returning the sequence and setting len on
    success, and returning NULL on failure. /
int *getValues( int *len ) {
  scanf( "%d", len );
  int *seq = (int *)malloc( *len * sizeof( int ) );
  for ( int i = 0; i < *len; i++ )
    if ( scanf( "%d", seq + i ) != 1 ) {
      free( seq );
      return NULL;
    }
  return seq;
}
```

What if this fails?

Island of Bad Programs

```
/** Read a sequence of values from the user,
    returning the sequence and setting len on
    success, and returning NULL on failure. /
int *getValues( int *len ) {
  if ( scanf( "%d", len ) != 1 )
    return NULL;
  int *seq = (int *)malloc( *len * sizeof( int ) );
  for ( int i = 0; i < *len; i++ )
    if ( scanf( "%d", seq + i ) != 1 ) {
      free( seq );
      return NULL;
    }
  return seq;
}
```

Fixed

Still something to think about. What if the user enters a really, really big number?

Island of Bad Programs

```
/** Read a sequence of values from the user,
    returning the sequence and setting len on
    success, and returning NULL on failure. /
int *getValues( int *len ) {
  if ( scanf( "%d", len ) != 1 )
    return NULL;
  int *seq = (int *)malloc( *len * sizeof( int ) );
  for ( int i = 0; i < *len; i++ )
    if ( scanf( "%d", seq + i ) != 1 ) {
      free( seq );
      return NULL;
    }
  return seq;
}
```

I'd call all of these **programming errors**.

Hurting Lots of People

- Should we care about this kind of error?
  - When a user crashes a program, aren't they just hurting themselves?
  - … well … no. Not necessarily.
- We will develop software that:
  - serves lots of users
  - runs with more privileges than its users
- Vulnerabilities can lead to:
  - Denial of service for lots of users
  - Exposure of secret information

More Tools for Worse Programs

- There's a powerful, dangerous function

```
int system( const char *command );
```

  - Give it a command, it will run it via the shell.
- This is a convenient, dangerous way to do lots of things you don't know how to do directly in C.

```
// I don't remember how to remove a file
system( "rm temp.txt" );

// I'm too lazy to write code to print out a file
system( "cat output.txt" );
```

More Tools for Worse Programs

- A function like system() is easy to use … and easy to abuse.
- How about this code to print out a user-selected file?

```
printf( “File to view: “ );
fgets( filename, sizeof(filename), stdin );

char cmd[ 1024 ] = “cat “;
strncat( cmd, filename, sizeof( cmd ) );

system( cmd );
```

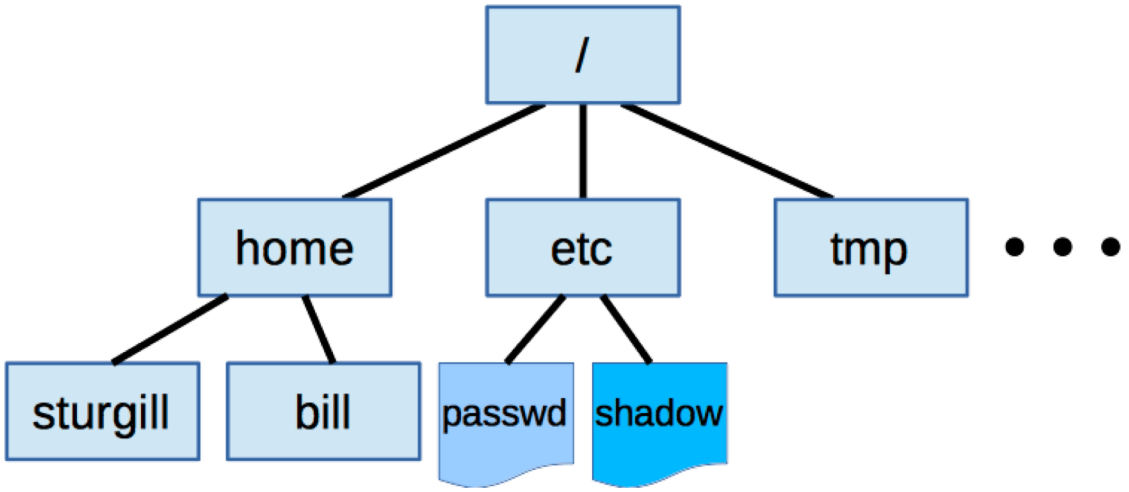
More Tools for Worse Programs

- A function like system() is easy to use … and easy to abuse.
- How about this code to print out a user-selected file?

```
printf( "File to view: " );
fgets( filename, sizeof(filename), stdin );

char cmd[ 1024 ] = "cat ";
strncat( cmd, filename, sizeof( cmd ) );

system( cmd );
```

That's an error right there, but it's not the worst one.

More Tools for Worse Programs

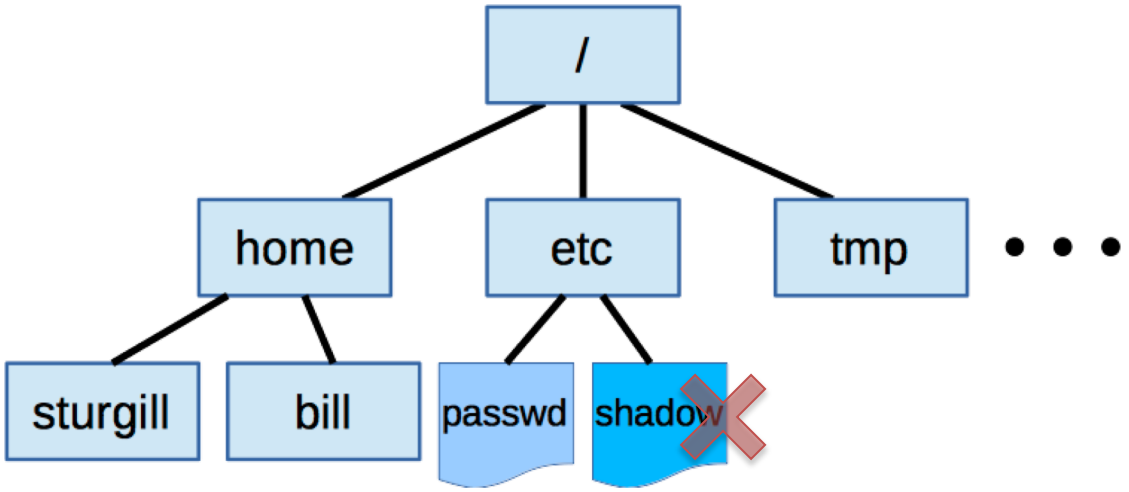



```
printf( "File to view: " );
fgets( filename, sizeof(filename), stdin );

char cmd[ 1024 ] = "cat ";
strncat( cmd, filename, sizeof(cmd)-strlen(cmd)-1 );

system( cmd );
```

If the user types **/etc/shadow**

cat /etc/shadow

More Tools for Worse Programs



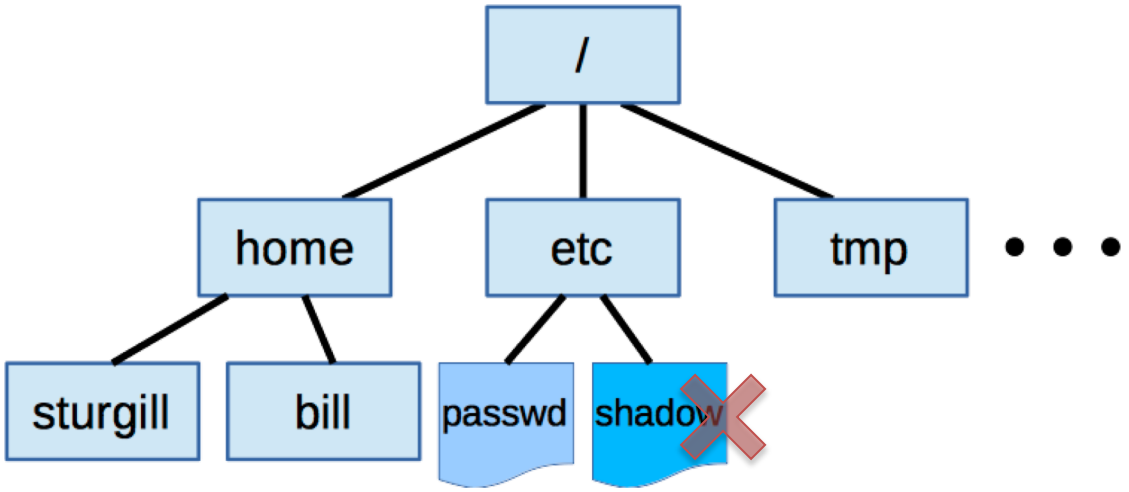
These are examples of
Path Manipulation

```
printf( "File to view: " );
fgets( filename, sizeof(filename), stdin );

char cmd[ 1024 ] = "cat ";
strncat( cmd, filename, sizeof( cmd ) );

system( cmd );
```

Or, worse, what if they type
a.txt; rm /etc/shadow

cat a.txt; rm /etc/shadow

More Tools for Worse Programs




```
printf( "File to view: " );
fgets( filename, sizeof(filename), stdin );

char cmd[ 1024 ] = "cat ";
strncat( cmd, filename, sizeof( cmd
                                    );

system( cmd );
```

I'd call this **failure to validate inputs**.

Validating Inputs

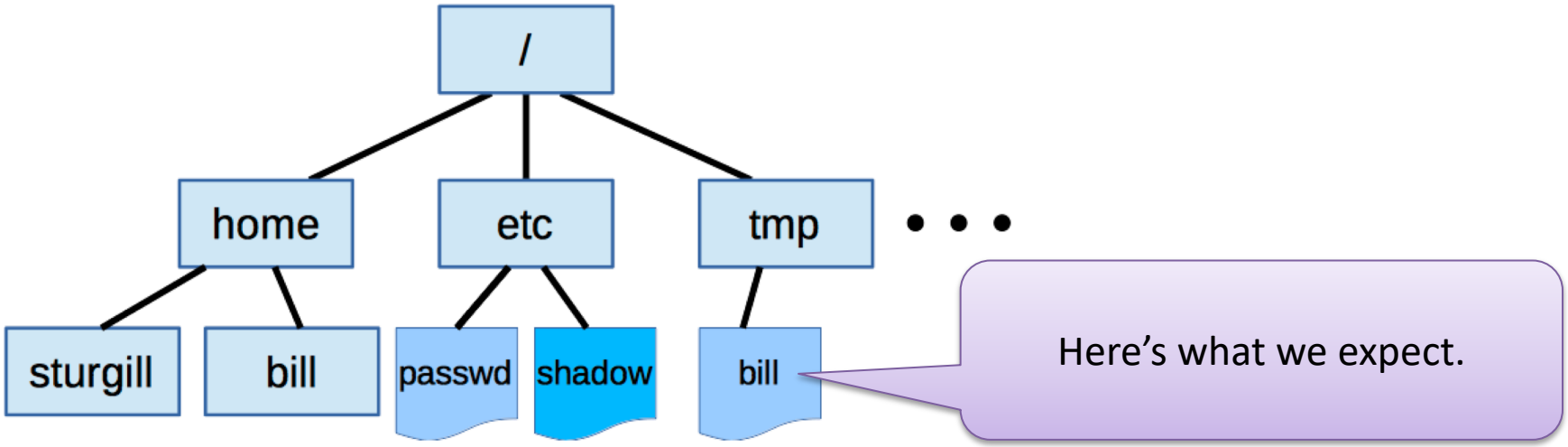
- Validate all inputs; don't rely on clients having done so
- Identify special (meta) characters and escape them consistently during input validation
- Use well-established, library functions to check for (a) legal URLs (b) legal filenames/pathnames (c) legal UTF-8 strings, …
- Use white listing (rather than black listing)
- Authenticate that the communication is from correct person

More Tools for Worse Programs

- A standard library function, getenv() will give us the value of environment variables.
- Useful, but not to be trusted.

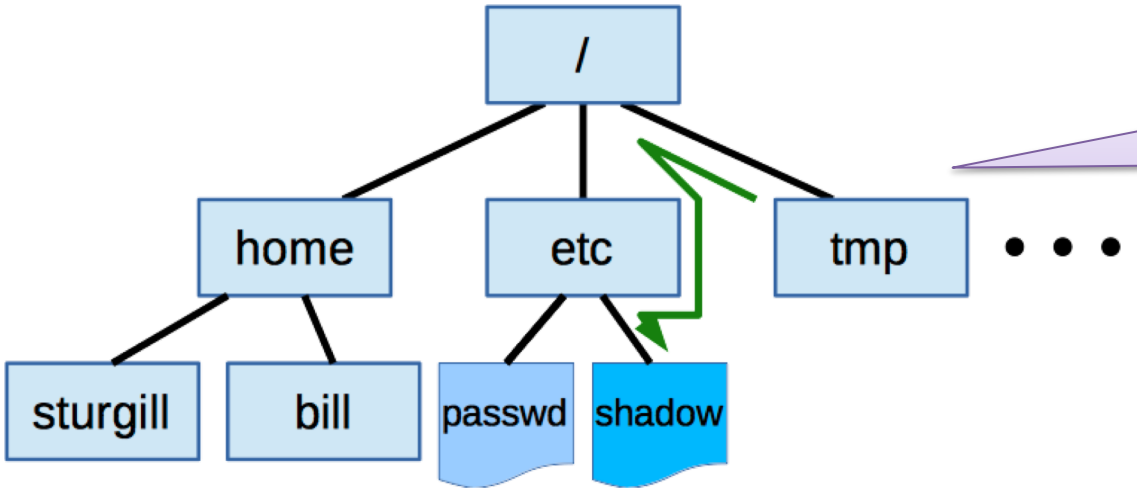
```
...
// Make a temporary file for this user.
char *user = getenv( “USER” );
char path[ 1024 ];
snprintf( path, sizeof(path), “/tmp/%s”, user );

FILE *fp = fopen( path, “w” );
...
```

More Tools for Worse Programs




```
...
// Make a temporary file for this user.
char *user = getenv( "USER" );
char path[ 1024 ];
snprintf( path, sizeof(path), "/tmp/%s", user );

FILE *fp = fopen( path, "w" );
...
```

More Fun with Path Manipulation



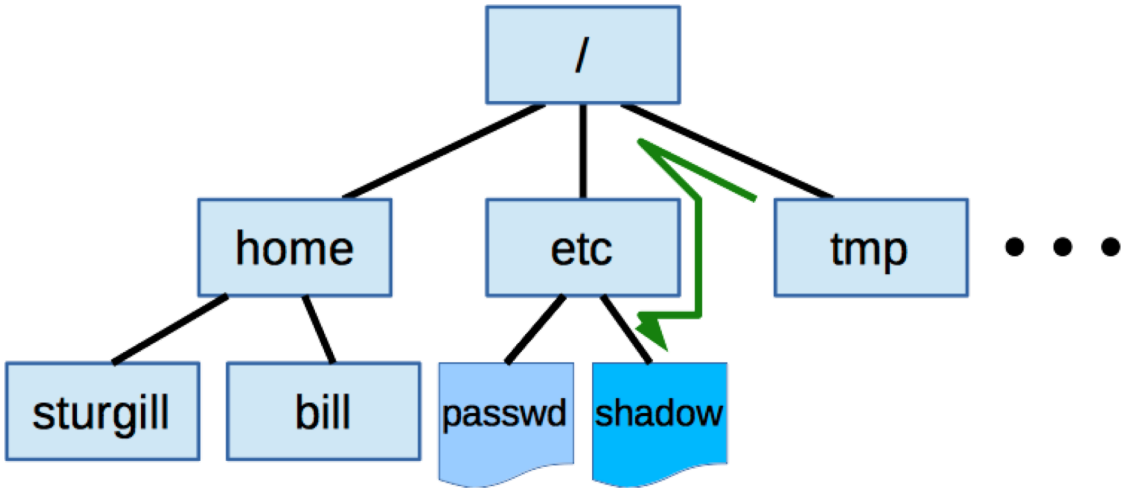
Bill could trick us into doing this.

```
...
// Make a temporary file for this user
char *user = getenv( "USER" );
char path[ 1024 ];
snprintf( path, sizeof(path), "/tmp/%s", user );

FILE *fp = fopen( path, "w" );
...
```

bill could do this:
$ USER=../etc/shadow

Overwrite
/tmp/../etc/shadow
Ouch!

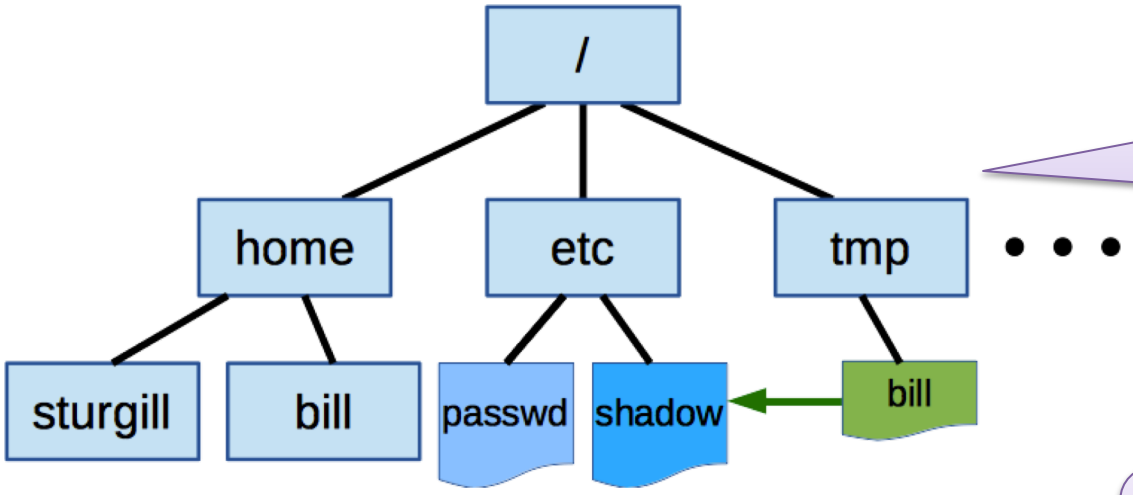
More Fun with Path Manipulation




```
...
// Make a temporary file for this user.
char *user = getenv( "USER" );
char path[ 1024 ];
snprintf( path, sizeof(path),

FILE *fp = fopen( path, "w" );
...
```

I'd call this **Inadequate Understanding of Operating Environment.**

More Fun with Path Manipulation



Any user who gets there first can do this.

```
...
// Make a temporary file for this u
char *user = getenv( "USER" );
char path[ 1024 ];
snprintf( path, sizeof(path), "/tmp/%s", user );

FILE *fp = fopen( path, "w" );
...
```

Any user can:
$ ln -s /etc/shadow /tmp/bill

Now, this will overwrite **/etc/shadow** via the symbolic link.

Logging

- Sensitive applications can use logs to record:
  - Startup and configuration actions
  - Important events
  - Errors that occur
- Structured logs can help us detect and diagnose problems afterwards
- But, manipulating logs offers a way to “sow confusion”

Inviting Log Forging

```
char str[ 1000 ];
int len;
printf( “cmd> “ );
if ( fgets( str, sizeof(str), fp ) ) {
   if ( valid( str ) ) {
     ...
   } else
     fprintf( log, “LOG: invalid string = %s\n”, str );
}
...
```

- Great. We can tell when something went wrong.
- … and maybe other nonsense we didn’t expect.

Log Forging

- If the a user enters:

```
cmd> complete-nonsense
```

- The log does its job:

```
LOG: invalid string = complete-nonsense
```

- But, an attacker may enter misleading strings:

```
cmd> complete-nonsense                                LOG: User
authenticated = Darth Vader
```

- This is going to create some misleading log entries:

```
LOG: invalid string = complete-nonsense
LOG: user authenticated = Darth Vader
```

Log Forging

- If the a user enters:

```
cmd> complete-nonsense
```

- The log does its job:

```
LOG: invalid string = complete-nonsense
```

- But, an attacker may enter misleading strings:

```
cmd> complete-nonsense
authenticated = Darth Vader
```

- This is going to create some misleading entries:

```
LOG: invalid string = complete-nonsense
LOG: user authenticated = Darth Vader
```

A bunch of spaces here.

Makes this look like a line break, if you're not careful.

Buffer Overflow … Revisited

- Buffer overflow will crash your program
  - If you're lucky.

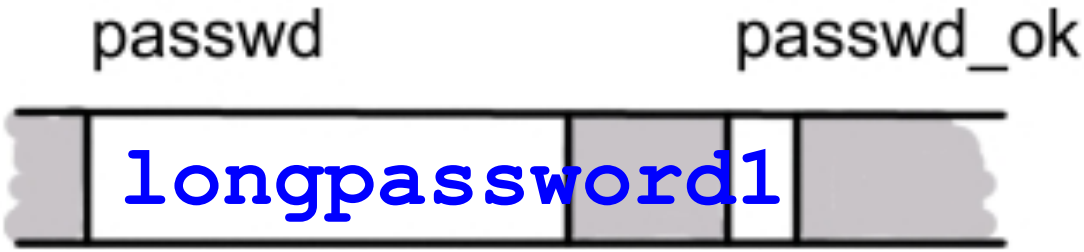
```
void f() {
  int a[10];
  a[20] = 3;
}
```

- If not, it's something an attacker can use to get your system
  - To make it fail in exactly the way she wants.

Example: Buffer Problem

```
bool passwd_ok = false;
char passwd[ 10 ];

printf( "Password: " );
scanf( "%s", passwd );

if ( strcmp( passwd, "joshua" ) == 0 )
  passwd_ok = true;

if ( passwd_ok ) {
  printf( "Welcome, the system is yours.\n" );
}
```

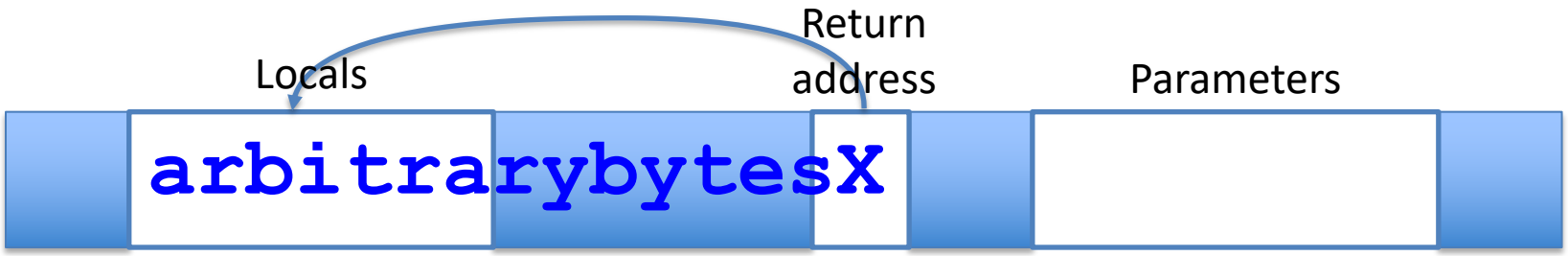
- Layout in memory:



- **`passwd`** buffer overflowed, overwriting **`passwd_ok`** flag
  - Any sufficiently long password will be accepted!

Stack Attacks

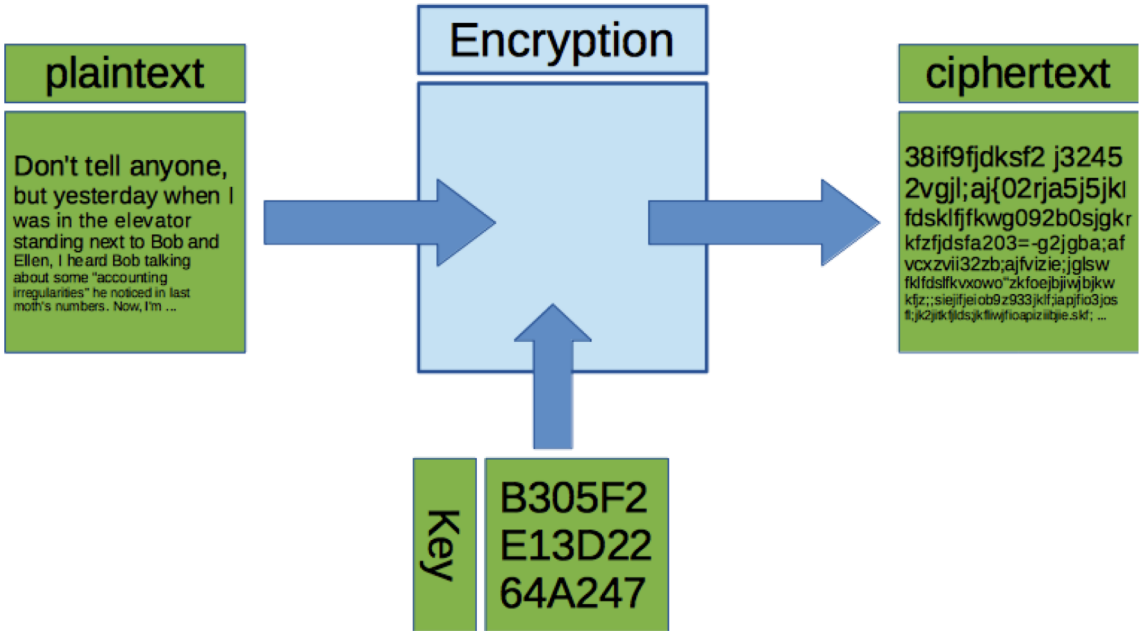
- When a function is called…
  - parameters are pushed on stack
  - return address pushed on stack
  - called function puts local variables on the stack
- Memory layout



- Problems?
  - Return to address X which may execute arbitrary code
  - Maybe even code they just wrote into memory.

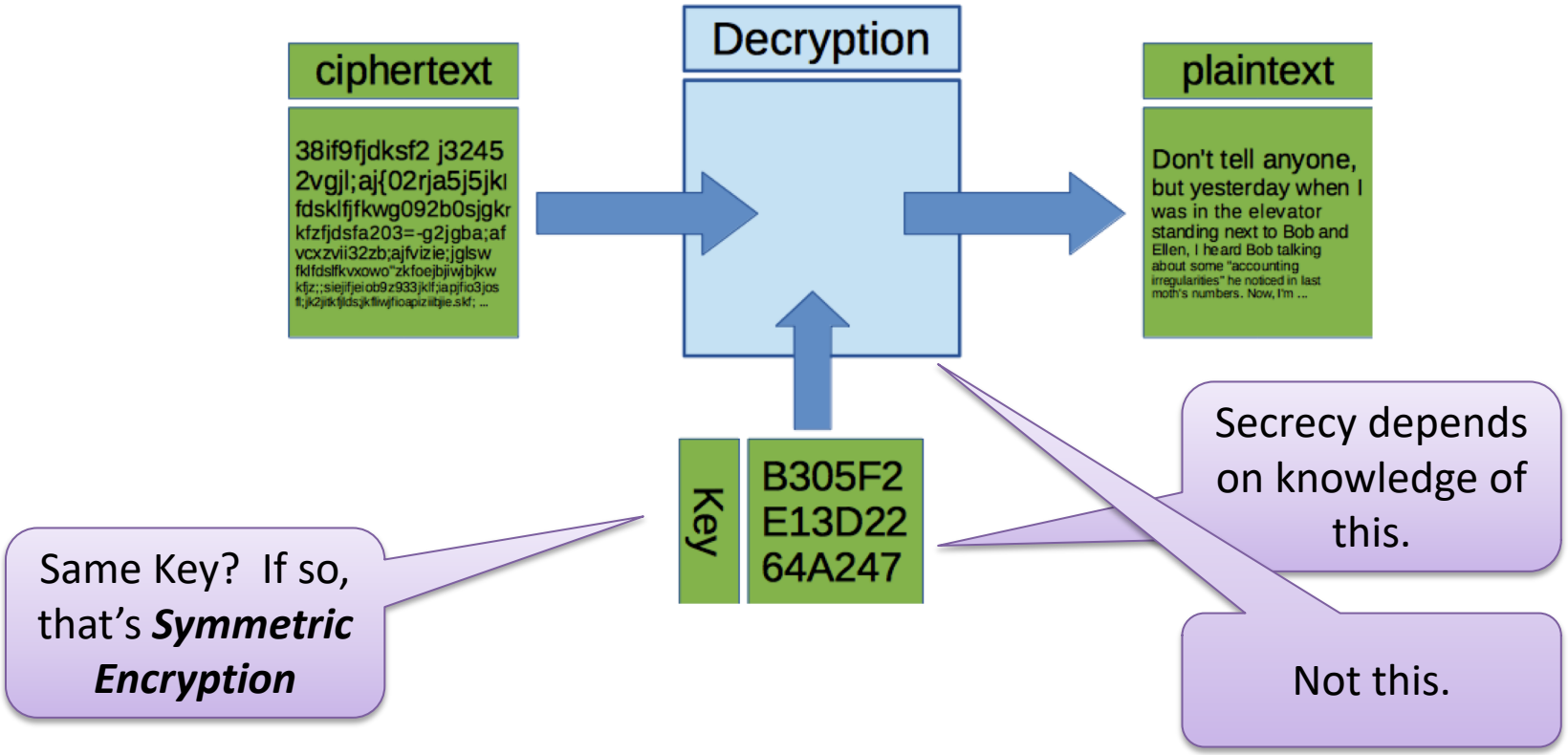
Encryption

- *Encryption* : Turning *plaintext* into something that looks like random garbage (ciphertext), using a *key*



Decryption

- Decryption: turning ciphertext back into plaintext, based on a key.



Symmetric Encryption Techniques

- There are a variety of symmetric encryption algorithms
  - DES : 56-bit key *block cipher.* Good enough … for 1977.
  - Many newer algorithms supporting a larger *keyspace*
    - 3DES, AES

Encrypts plaintext in fixed-sized blocks

Fun with Encryption

- The openssl command knows a lot of encryption techniques

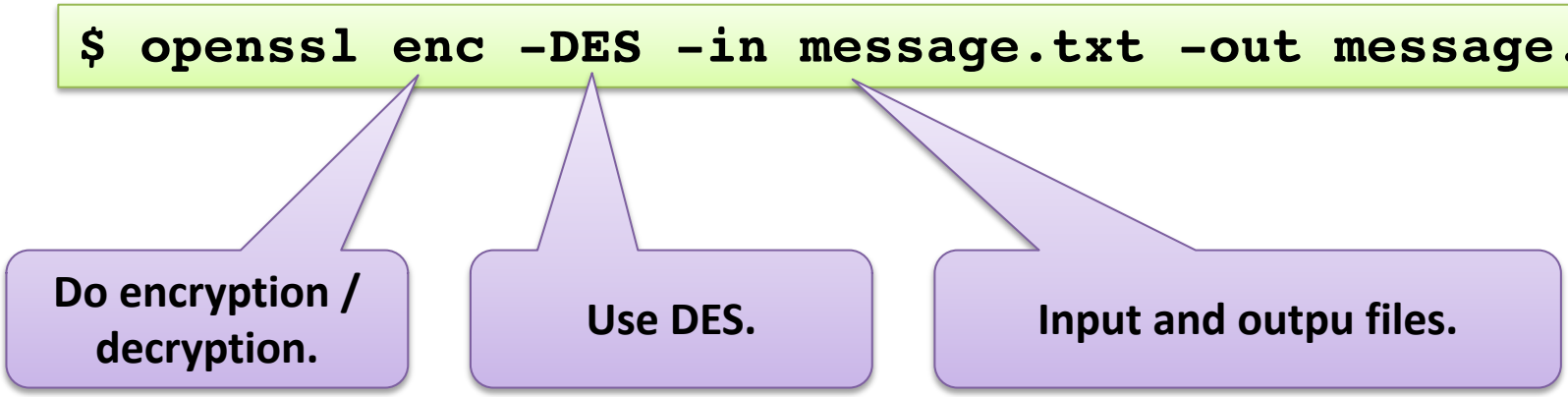
```
$ openssl list-cipher-algorithms
AES-128-CBC
AES-128-CBC-HMAC-SHA1
... Big list of encryption techniques.
DES => DES-CBC
DES-CBC
... More techniques
AES-128-CBC
... Even more
```

- It’s a command-line utility that implements the Secure Sockets Layer and related protocols

Fun with Encryption

- Let's give it a try:
  - Make a file message.txt containing anything you want.
  - Encrypt the file with openssl

```
$ openssl enc -DES -in message.txt -out message.bin
```



Fun with Encryption

- We get a new, encrypted file, message.bin
  - The original file is still there.
  - Are you brave enough to delete it? I am.

```
$ rm message.txt
```

- I hope I can get it back. Of course, that's what encryption is all about.

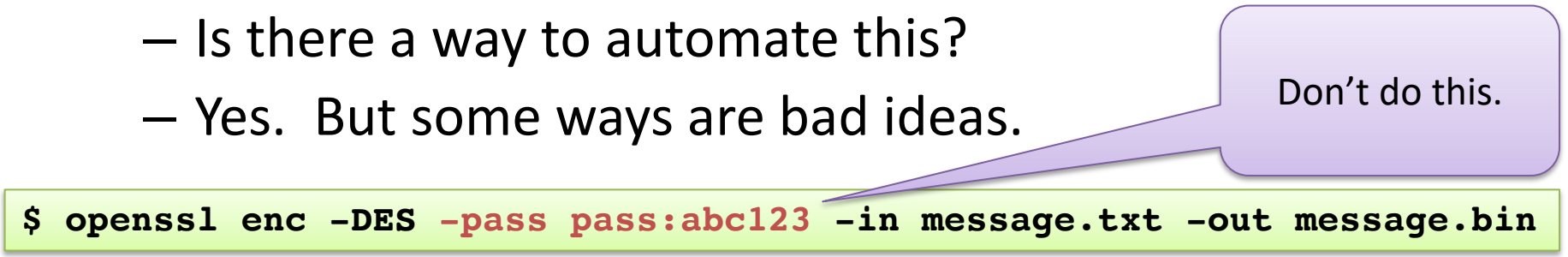
```
$ openssl enc -DES -d -in message.bin -out message.txt
```

The -d is for decrypt.

- I have to provide the same key for this to work.

A Bad Idea

- I don't like typing in my password twice.
  - Is there a way to automate this?
  - Yes. But some ways are bad ideas.



- Anyone on the system can potentially see this password
- This is really about **knowing your execution environment.**

Encryption from C

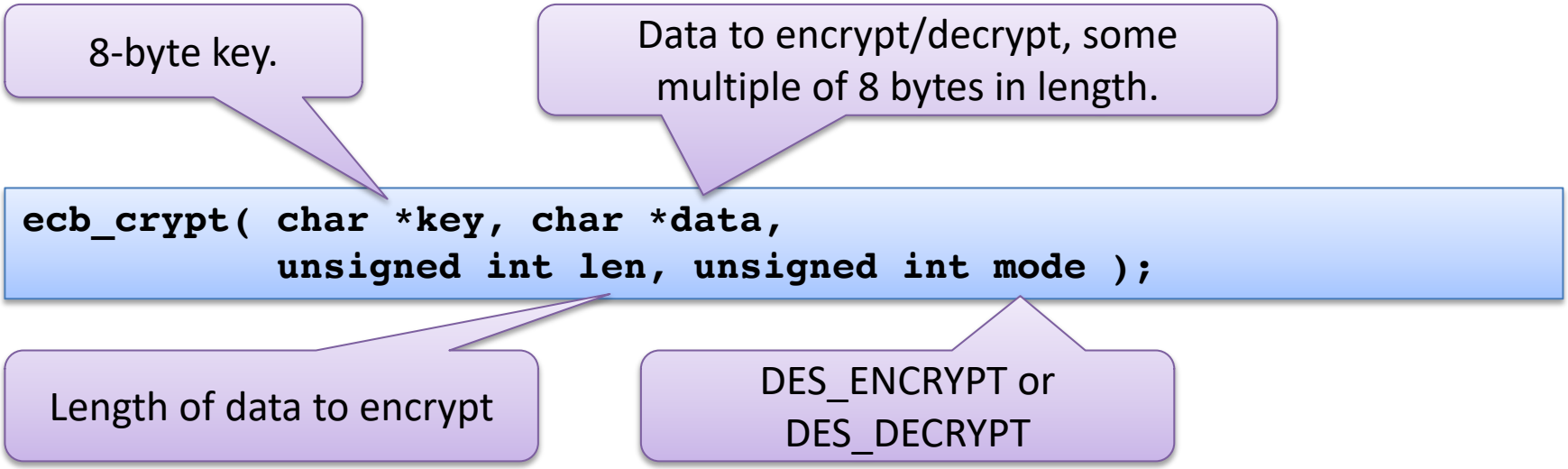
- We could run a command like this using system()

```
system( "openssl enc -DES -d -pass pass:abc123 ..." );
```

- But, that might expose our password
  - … and it could be slow.

Encryption from C

- There are libraries to encrypt / decrypt for you.
- libcrypt is available on many Unix/Linux systems (although it's not part of the C standard)
- You have to include the **rpc/des_crypt.h** header.
- This gives you a few functions:



Key Preparation

- The key has a parity bit for each byte.
  - Before you can use it, you need to set up its parity bits.
  - There's a function that will do this for you.

```
des_setparity( key );
```

8-byte key

Encrypt / Decrypt Example

```
#include <rpc/des_crypt.h>
```

```
char key[] = "abcd1234";
des_setparity( key );


char message[] = "Here, an example message";
int msgLen = strlen( message );



ecb_crypt( key, message, msgLen, DES_ENCRYPT );



ecb_crypt( key, message, msgLen, DES_DECRYPT );
```

Prepare the 8-byte key.

Make a 24-byte message.

Encrypt the message (in place)

Decrypt the message (in place)

Linking with libcrypt

- These functions are in a library, libcrypt
- To build an executable, you need to link with this library.

```
$ gcc -Wall -std=c99 -g des.c -o des -lcrypt
```

Link with libcrypt

Actually, my executable compiled OK without this.

Software Security

- Think about security up-front
- Consider security as functionality rather than hidden part of system
- Design and test with security in mind
- Protect your secrets and paths of communication
  - Cryptography
- Program defensively
  - Validate inputs
  - Check buffers and bounds
- Verification and Validation
  - Test! Think maliciously! How could you attack a system?
  - Use tools that support identifying security vulnerabilities.

Hiding Information

- It can be hard to hide information in a program:

```
char passwd[ 64 ];
printf( “Enter password: “ );
scanf( “%63s”, passwd );
if ( strcmp( passwd, “Albuquerque” ) == 0 ) {
    ...
}
```

- Not very secure if this is open source.
- But, even if it’s not …

```
$ strings program
Enter password:
%63s
Albuquerque
...
```

Hiding Information

- It can be hard to hide information in a program:

```
char passwd[ 64 ];
printf( "Enter password: " );
scanf( "%63s", passwd );
if ( strcmp( passwd, "Albuquerque" ) == 0 ) {
    ...
}
```

- Not very secure if this is open source.
- But, even if it's not …

```
$ strings program
Enter password:
%63s
Albuquerque
...
```

I'd call this **Inadequate Protection of Secret Information.**

Depending on Randomness

- We don't usually depend on hiding secrets in an executable
  - Usually, we want to choose our own secrets
  - … and store them as part of our configuration
- This can benefit from a source of randomness
  - How about this:

```
char passwd[ 15 ];
for ( int i = 0; i < sizeof( passwd ) - 1; i++ )
  passwd[ i ] = 'a' + rand() % 26;
passwd[ sizeof( passwd ) - 1 ] = '\0';
```

Depending on Randomness

- Bad idea. This is a pseudo-random sequence. It's the same from execution to execution.
- How about this instead?

```
srand( time( NULL ) );
char passwd[ 15 ];
for ( int i = 0; i < sizeof( passwd ); i++ )
  passwd[ i ] = 'a' + rand() % 26;
passwd[ sizeof( passwd ) - 1 ] = '\0';
```

- Not much better. The space of possible times is still quite small.

More Random Randomness

- Unix-like systems offer /dev/random
  - A sequence of random bytes
  - Based on pseudorandom processes and environmental noise
- Need 16 bytes of randomness? No problem.

```
$ dd count=16 bs=1 if=/dev/random of=randbits.bin
```

- Random enough for you?

```
$ hexdump -C randbits.bin
00000000  de 92 b9 5e 52 75 e3 0b  f3 50 29 20 83 3b ac 86
```