CSci 4271W
Development of Secure Software Systems
Day 8: Defensive programming 1

Stephen McCamant (he/him)
University of Minnesota, Computer Science & Engineering

Based in large part on slides originally by Prof. Nick Hopper
Licensed under Creative Commons Attribution-ShareAlike 4.0

Last time: bugs and attacks

```
void checkpassword(FILE *pwfile) {
  int taunt = 1;
  char password[10], input[10];
  char *inp = input;
  fgets(password,9,pwfile);
  password[8]='\0';
  printf("Enter password (at most 8 letters):");
  do {
    *inp = getchar();
  } while (*inp++ != '\n');
  input[8] = '\0';
  if (strncmp(input,password,8) == 0) taunt = 0;
  if (taunt) {
    printf("Loser, the password is definitely not ");
    printf(input);
  } else return success();
}
```

Outline

Carry-over from last lecture

Defensive coding

Announcements intermission

Defensive coding, cont'd

Defensive programming

How do we design and write programs so that security bugs are less likely to happen in the first place?
Like in threat modeling, it's important to ask "what could go wrong" in a program and write code that plans for it.

- Undefined behavior: what can go wrong with the compiler?
- Input validation: what can go wrong with our inputs?
- Safe call-outs: what can go wrong when calling out?
- Safe return values: what can go wrong with my outputs?

Undefined behavior

In C/C++, some operations have undefined behavior: an execution that would lead to one of these operations can behave arbitrarily. Examples include:

- Dereferencing a null pointer
- Reading uninitialized memory (stack, `malloc`, fields...)
- Signed integer overflow, including shifts and casts
- Using non-aligned or dangling pointers
- Void return from non-void function

Null pointers

Refresher: which of these is a null pointer dereference?

```
char *x = NULL;
*x = '1';
```

```
void *p = NULL;
void **q = &p;
```

```
char *x = NULL;
if (x==NULL) return;
```

```
std::string *s = NULL;
s->push_back('a');
```

```
int *x = NULL;
x[0] = 42;
```

```
char *x = NULL;
if (!x) return;
```

(Signed) integer overflow

...is tricky to test for without incurring undefined behavior:

```
int x, y; // ...
if (x>=0 && y>=0 && x+y < x)
   // error...
```

(Signed) integer overflow

...is tricky to test for without incurring undefined behavior:

```
int x, y; // ...
if (x>=0 && y>=0 && x+y < x)
   // error...
```

(Signed) integer overflow

...is tricky to test for without incurring undefined behavior:

```
int x, y; // ...
if (x>=0 && y>=0 && x+y < 0)
   // error...
```

```
int x, y;
// Want to know if x+y > INT_MAX
if (x>=0 && y>=0 && x > (INT_MAX - y))
   // error...
```

Consider compiling with `-fsanitize=integer`

Non-defensive example 1

```
char *double_str (char *s) {
  int len = strlen(s);
  char *p = malloc(2*len+1);
  strcpy(p,s);
  strcpy(p+len,s);
  return p;
}
```

Input validation

- Check all inputs for safe/sane values.
- Default to reject/deny, and only allow known-safe values.
- Do test with known-bad values!
  - Integer types: 0, 1, -1, `INT_MAX`, `INT_MIN`, ...
  - Strings: NULL, `""`, non-NUL terminated, long strings, unprintable characters, newlines, %n, ...
  - Fuzz your interface to find other cases

Input sources

- Command line inputs
- Function arguments / global variables
- File contents
- Environment variables
- Network / IPC

(Wheeler has good list of considerations for common types)

Non-defensive example 2

```
int main(int argc, char **argv) {
  string greeting("Hello ");
  if (!strcmp(argv[0],"hello")) greeting.append(getenv("USER"));
  else
    cin >> greeting;
  int nope = greeting.size();
  if (greeting.find('|') == nope && greeting.find(';') == nope){
    greeting.insert(0,"/bin/echo ");
    system(greeting.data());
  } else
    cout << "You can't fool me!" << endl;
  return 0;
}
```

Outline

Important dates next week

- Homework 2 is due next Tuesday the 18th
  - Today's defensive programming material should prepare your for questions 2–4
  - Homework 1 grades are just waiting for my final checks
- Midterm 1 will be in class next Thursday the 20th
  - Open book, open notes, questions similar to homework and short answers
  - More detailed discussion of the midterm on Tuesday

Project 1 starting up

- Instructions on the public web site, vulnerable code on `github.umn`
- The project is time-consuming so get started early
  - Groups of 2 or 3 recommended
- One section draft per student will be due 3/4
- Final due date is after spring break, 3/18

Outline

Carry-over from last lecture

Defensive coding

Announcements intermission

Defensive coding, cont'd

Safe calling (programs)

When calling out to another program:

- Use the program's full path
- Watch out for "metacharacters" that are special to the shell
  - Avoiding the shell avoids this risk, but is cumbersome and less portable
- Watch out for characters the program will interpret differently
- Avoid interactive programs
- Results from the program are inputs to yours: validate!

Safe calling (functions)

System and library calls: check results!

- Calls that return pointers: check for `NULL`
- Check error codes for system calls:
  - `open`, `write`, `read`...
- Check `errno` for calls that set it
- Catch exceptions

Use "safer" libraries/APIs where you can: C++ `string` vs. `char *`, well-tested libraries, standard collections, ...

Non-defensive example 3

```
char *username = getenv("USER");
char *buf = malloc(strlen(username)+7);
sprintf(buf, "mail %s", username);
FILE *f = popen(buf, "w");
fprintf(f, "Hi.\n");
fclose(f);
```

Outputs

The outputs of a program can potentially cause information disclosure, sometimes leading to EoP.

- "The third letter of your password is wrong"
- Query FAILED with output...

Feedback and errors

Examples that may reveal too much information:

- Displaying incorrect password/secret
- Returning program line/check causing an error
- Returning "hidden" information in comments or fields
- Returning error message from a library or external program

Non-defensive example 4

```
try:
  connect_to_db(dbuser, dbpassword)
except Exception as e:
  return str(e)

if user not in userlist:
  return "User not found!"
if hashlib.sha256(password) != pw_hash_list[user]:
  return "Incorrect password!"
if account_number not in accounts[user]:
  return "Account number " + account_number + " not found!"

do_the_thing(user,account)
```

Formatting and encoding

- Will the consumer of output be another program?
  - What text encoding should be used?
  - Are tags, control characters, etc., important?
- Don't use user-specified formats
  - Especially not format strings
  - Limited choices can be safe