Lec 4: Secure Coding

CS492E: Introduction to Software Security

Sang Kil Cha

Writing Secure Code

Software Bug = Root of Evil

Bugs can be exploited by an attacker to compromise the entire system.

Our Goal

Error-free Software!

Defensive Programming

Making the software behave in a predictable manner despite unexpected inputs or user actions[1].

Secure coding is a type of defensive programming that mainly concerns with computer security.

[1] https://en.wikipedia.org/wiki/Defensive_programming

Insecure vs. Secure

```c
int func(char * input)
{
  char buf[8];
  strcpy(buf, input);
  // ...
}
```

```c
int func(char * input)
{
  char buf[8];
  strncpy(buf, input, 8);
  buf[7] = 0;
  // ...
}
```

Problem of Defensive Programming

```
int x = 1;
int y = 2;
int z = x + y;
if ( z > x && z > y ) return z;
else abort();
```

Problem of Defensive Programming

```
int x = 1;
int y = 2;
int z = x + y;
if ( z > x && z > y ) return z;
else abort();
```

Defensive programming can

- introduce redundancy.

Problem of Defensive Programming

```
int x = 1;
int y = 2;
int z = x + y;
if ( z > x && z > y ) return z;
else abort();
```

Defensive programming can

- introduce redundancy.
- introduce hard-to-read code.

Problem of Defensive Programming

```
int x = 1;
int y = 2;
int z = x + y;
if ( z > x && z > y ) return z;
else abort();
```

Defensive programming can

- introduce redundancy.
- introduce hard-to-read code.
- slow down the program.

Offensive Programming

- A category of defensive programming (***not the opposite***).
- Make defensive checks to be unnecessary by failing fast.

Defensive vs. Offensive Programming

```
void addElement(list<int>* lst, int el) {
  if (lst != NULL) {
    lst->push_back(el);
  }
}
```

vs.

```
void addElement(list<int>* lst, int el) {
  assert(lst); // fail-fast!
  lst->push_back(el);
}
```

Where to Put Guards?

Defensive/offensive programming is a good start, but you should really know ***where*** to put your guards.

Secure Coding Guideline

SEI CERT C Coding Standard

- https://www.securecoding.cert.org/confluence/display/c/SEI+CERT+C+Coding+Standard
- Similar guidelines available for other languages, too.
- Quite a lot of rules.
- You should really read through them all!
- We will discuss only a few of them in this lecture.

Rule #1: Declare Objects with Appropriate Storage Duration

In the C Standard, 6.2.4, paragraph 2 [ISO/IEC 9899:2011]

The lifetime of an object is the portion of program execution during which storage is guaranteed to be reserved for it. An object exists, has a constant address, and retains its last-stored value throughout its lifetime. ***If an object is referred to outside of its lifetime, the behavior is undefined***. The value of a pointer becomes indeterminate when the object it points to reaches the end of its lifetime.

Example 1

```
const char *p;
void dont_do_this(void) {
  const char c_str[] = "This␣will␣change";
  p = c_str;
}
void innocuous(void) {
  printf("%s\n", p);
}
```

Example 2

```
char *init_array(void) {
  char array[10];  /* Initialize array */
  return array;
}
```

Rule #2: Avoid Linkage Conflict!

What is a linkage in C?

Linkage in C

- Any identifiers can have a linkage, which is an attribute that describes whether two or more declarations of the same identifier refer to the same entity or not.
- Three kinds:
  - ***No linkage***: the identifier can only be referred within its scope.
  - ***Internal linkage***: the identifier can be referred to from all scopes in the current translation unit[2].
  - ***External linkage***: the identifier can be referred to from all scopes in other translation units.

[2] Roughly, a translation unit includes the current source file and header files included by it.

When an Identifier Declared Twice

		Second		
		`static`	no linkage	`extern`
First	`static`	internal	***undefined***	internal
	no linkage	***undefined***	no linkage	external
	`extern`	***undefined***	***undefined***	external

Example 3

```
int i1 = 10;
static int i2 = 20;
extern int i3 = 30;
int i4;
static int i5;

int i1;  /* Valid */
int i2;  /* Undefined */
int i3;  /* Valid */
int i4;  /* Valid */
int i5;  /* Undefined */
```

This code compiles on MSVC

Rule #3: Do Not Depend on the Order of Evaluation for Side Effects

```
i = i + 1;     // okay
i = ++i + 1;   // not okay

a[i] = i;      // okay
a[i++] = i;    // not okay
```

Example 4

```
void func1(int i, int *b) {
  int a = i + b[++i];
  printf("%d, %d", a, i);
}

void func2() {
  foo(i++, i);
}
```

Example 5

```
// The order of evaluation for function arguments is
// unspecified.
extern void c(int i, int j);
int glob;
int a() { return glob + 10; }
int b() { glob = 42; return glob; }

void func() { c( a(), b() ); }
```

Rule #4: Do Not Read Uninitialized Memory

```
void set_flag(int number, int *sign_flag) {
  if (NULL == sign_flag) return;
  if (number > 0) {
    *sign_flag = 1;
  } else if (number < 0) {
    *sign_flag = -1;
  }
}
int is_negative(int number) {
  int sign;
  set_flag(number, &sign);
  return sign < 0;
}
```

Example 6 (CVE-2009-1888)

```
void func(const char *mbs) {
  size_t len;
  mbstate_t state;

  len = mbrlen(mbs, strlen(mbs), &state);
}
```

Example 6 (CVE-2009-1888)

```
void func(const char *mbs) {
  size_t len;
  mbstate_t state;

  len = mbrlen(mbs, strlen(mbs), &state);
}
```

`mbrlen` updates the shift state (third argument).

Rule #5: Do Not Dereference NULL

In computing, a null pointer or null reference is a value saved for indicating that the pointer or reference does not refer to a valid object[3].

```
int *p = NULL;
// ...
*p = 42; // ?
```

[3] https://en.wikipedia.org/wiki/Null_pointer

Example 7

```
void
func(png_structp png_ptr, int length, const void *user_data)
{
  png_charp chunkdata;
  chunkdata = (png_charp)png_malloc(png_ptr, length + 1);
  /* ... */
  memcpy(chunkdata, user_data, length);
  /* ... */
}
```

Variadic Function?

A function of indefinite arity, i.e., a function with an argument "..." (variable number of args). The following four macros used to access variables.

- `va_list`: a type for variadic function's arguments.
- `va_start(a, count)`: a function that initializes a va_list object.
- `va_end(a)`: a function that finalizes a va_list object.
- `va_arg(a, type)`: returns the next arg of the specified type.

Variadic Example

```
double average(int count, ...)
{
  va_list ap;
  int j;
  double sum = 0;
  va_start(ap, count);
  for (j = 0; j < count; j++) {
    sum += va_arg(ap, int);
  }
  va_end(ap);
  return sum / count;
}
```

Rule #6: Do Not Call `va_arg` with an Argument of the Incorrect Type

```
void func(size_t count, ...) {
  va_list ap;
  va_start(ap, count);
  if (count > 0) {
    unsigned char c = va_arg(ap, unsigned char);
    // ...
  }
  va_end(ap);
}
void f() {
  unsigned char c = 0x12;
  func(1, c);
}
```

Default Promotion

```
printf("%c, %d", (char) 1, (int) 42)
```

- Any integer type less than `int` is promoted to `int`.
  - So the `char` becomes `int`.
- Float is promoted to `double`.

Example Revisited

```
void func(size_t count, ...) {
  va_list ap;
  va_start(ap, count);
  if (count > 0) {
    unsigned char c = va_arg(ap, unsigned char); // how to fix?
    // unsigned char c = (unsigned char) va_arg(ap, int);
    // ...
  }
  va_end(ap);
}
void f() {
  unsigned char c = 0x12;
  func(1, c); // what's the type of c?
}
```

Rule #7: Ensure that Signed/Unsigned Integer Operations Do Not Wrap

```
int x = 2147483647;
int y = 1;
printf("%d\n", x + y); // ?
```

Example 8 (CVE-2014-4377)

```
#include <limits.h>

void func(unsigned int ui_a, unsigned int ui_b) {
  unsigned int usum;



    usum = ui_a + ui_b;

    /* ... */
}
```

Example 8 (CVE-2014-4377)

```c
#include <limits.h>

void func(unsigned int ui_a, unsigned int ui_b) {
  unsigned int usum;
  if (UINT_MAX - ui_a < ui_b) {
    /* Handle error */
  } else {
    usum = ui_a + ui_b;
  }
    /* ... */
}
```

Rule #8: Ensure that Integer Conversions do Not Result in Lost of Misinterpreted Data

```
int x = 0x10000000;
char c = (char) x;
```

What should be the expected behavior?

Example 9

```
void func() {
  unsigned long int u_a = ULONG_MAX;
  signed char sc;
  sc = (signed char)u_a;
  /* ... */
}
```

Rule #9: Avoid Divide-by-Zero

When there is a divide by zero, CPU interrupts and program terminates because ALU cannot handle it: we don't know how to represent "infinity".

Example 10

```c
#include <limits.h>

void func(signed long s_a, signed long s_b) {
  signed long result;
  if ((s_a == LONG_MIN) && (s_b == -1)) { // how to fix this?
    /* Handle error */
  } else {
    result = s_a / s_b;
  }
  /* ... */
}
```

Rule #10: Avoid Using Floating-Point Variables as Loop Counters

The precision differs depending on the CPU!

A famous example in Python:

```
>>> 3 * 0.1
0.30000000000000004
```

Example 11

```
void func() {
  for (float x = 0.1f; x <= 1.0f; x += 0.1f) {
    /* Loop may iterate 9 or 10 times */
  }
}

void func() {
  for (float x = 100000001.0f;
       x <= 100000010.0f; x += 1.0f) {
    /* Loop may not terminate */
  }
}
```

Rule #11: Do Not Form or Use Out-of-Bounds Pointers or Array Subscripts

```
enum { TABLESIZE = 100 };

static int table[TABLESIZE];

int *f(int index) {
  if (index < TABLESIZE) { // Is this check safe?
    return table + index;
  }
  return NULL;
}
```

Rule #12: Guarantee that Library Functions Do Not Form Invalid Pointers

```
#include <string.h>
#include <wchar.h>

static const char str[] = "Hello world";
static const wchar_t w_str[] = L"Hello world";
void func() {
  char buffer[32];
  wchar_t w_buffer[32];
  memcpy(buffer, str, sizeof(str)); /* Compliant */
  wmemcpy(w_buffer, w_str, sizeof(w_str)); /* Noncompliant */
}
```

Rule #13: Guarantee that Storage for Strings has Sufficient Space

```
void copy(size_t n, char* src, char* dest) {
    size_t i;

    for (i = 0; src[i] && (i < n); ++i) {
      dest[i] = src[i];
    }
    dest[i] = '\0'; // ?
}
```

Rule #14: Do Not Pass a Non-Null-Terminated Character Sequence to a String Argument

```
void func() {
  char c_str[3] = "abc";
  printf("%s\n", c_str);
}
```

Rule #15: Do Not Access Freed Memory

```
struct node {
  int value;
  struct node *next;
};

void free_list(struct node *head) {
  for (struct node *p = head; p != NULL; p = p->next) {
    free(p); // how should we fix this?
  }
}
```

Fix

```
void free_list(struct node *head) {
  struct node *q;
  for (struct node *p = head; p != NULL; p = q) {
    q = p->next;
    free(p);
  }
}
```

Example 12

```
void f(char *c_str1, size_t size) {
  char *c_str2 = (char *)realloc(c_str1, size);
  if (c_str2 == NULL) {
    free(c_str1); // ?
  }
}
```

What if the second argument to `realloc` is `NULL`? Read the manual of `realloc`!

Example 13

```
enum { SIG_DESC_SIZE = 32 };

typedef struct {
  char sig_desc[SIG_DESC_SIZE];
} signal_info;

void func(size_t num_of_records,
          size_t temp_num,
          const char *tmp2,
          size_t tmp2_size_bytes) {
  signal_info *start =
   (signal_info *)
   calloc(num_of_records,
          sizeof(signal_info));
  if (tmp2 == NULL) { /* Handle Err */ }
  else if (temp_num > num_of_records)
  { /* Handle error */ }
  else if (tmp2_size_bytes < SIG_DESC_SIZE) {
    /* Handle error */
  }

  signal_info *point = start + temp_num - 1;
  memcpy(point->sig_desc, tmp2,
         SIG_DESC_SIZE);
  point->sig_desc[SIG_DESC_SIZE - 1] = '\0';
  /* ... */
  free(start);
}
```

Example 13

```
enum { SIG_DESC_SIZE = 32 };

typedef struct {
  char sig_desc[SIG_DESC_SIZE];
} signal_info;

void func(size_t num_of_records,
          size_t temp_num,
          const char *tmp2,
          size_t tmp2_size_bytes) {
  signal_info *start =
   (signal_info *)
   calloc(num_of_records,
          sizeof(signal_info));
  if (tmp2 == NULL) { /* Handle Err */ }
  else if (temp_num > num_of_records)
  { /* Handle error */ }
  else if (tmp2_size_bytes < SIG_DESC_SIZE) {
    /* Handle error */
  }

  signal_info *point = start + temp_num - 1;
  memcpy(point->sig_desc, tmp2,
         SIG_DESC_SIZE);
  point->sig_desc[SIG_DESC_SIZE - 1] = '\0';
  /* ... */
  free(start);
}
```

What if `calloc` fails?

And Many More ...

Check out the website: `https://www.securecoding.cert.org/confluence/display/c/SEI+CERT+C+Coding+Standard`

Conclusion

- There are so many undefined cases.
- Each of such cases can lead to an attack as we will see throughout the course.

Question?