CS412 Software Security

Software Bugs

Mathias Payer

EPFL, Spring 2019

From Software Bugs to Attack Primitives

- *Attack primitives* are exploit building blocks
- *Software bugs* map to *attack primitives*, i.e., enable computation
- A chain of *attack primitives* results in an *exploit*, the underlying bugs of the attack primitives become *vulnerabilities*

Attack primitive: arbitrary write

```c
int global[10];

void set(int idx, int val) {
  global[idx] = val;
}
```

An attacker with control of `idx` and `val` can set any 4b location +/- 2GB around global to an arbitrary value.

Attack primitive: arbitrary write, limited location

```c
void vuln(char *u1) {
  /* assert(strlen(u1) < MAX); */
  char tmp[MAX];
  strcpy(tmp, u1);
  /* equivalent:
     while (*u1 != 0)
       *(tmp++) = *u1++;
   */
  return strcmp(tmp, "foo");
}
```

An attacker with control of `u1` can overwrite values (except \0) on the stack above `tmp`, ending with an \0 byte.
Note that constrained writes only allow some values to be written.

Attack primitive: arbitrary read

```c
int global[10];

int get(int idx) {
  return global[idx];
}
```

An attacker with control over `idx` and the return value can read arbitrary 4b values +/- 2 GB of global's address.

Common bug types

Not all bugs map as clearly to primitives as the earlier examples. C/C++ provides many different opportunities for failure.

Improper initialization

```
typedef unsigned int uint;
int getmin(int *arr, uint len) {
  int min;
  for (int i=0; i<len; i++)
    min = (min < arr[i]) ? min : arr[i];
  return min;
}
```

Improper initialization

```
typedef unsigned int uint;
int getmin(int *arr, uint len) {
  int min;
  for (int i=0; i<len; i++)
    min = (min < arr[i]) ? min : arr[i];
  return min;
}
```

min is not initialized and may have an arbitrary value.

Side effects

```
if (foo == 12 || (bar = 13))
  baz == 12;
```

Side effects

```c
if (foo == 12 || (bar = 13))
  baz == 12;
```

`bar` is set if `foo!=12`, while `baz` is never set. Watch out when calling functions in an expression, their side effects will linger.

Scoping

```
int a;
void calc(int b) {
  int a = b*12;
  if (b + 24 == 96)
    a = b;
}
```

Scoping

```
int a;
void calc(int b) {
  int a = b*12;
  if (b + 24 == 96)
    a = b;
}
```

The local variable a is assigned while the global variable a is not modified.

Operator precedence

```
node *find(node **curr, val) {
  while (*curr != NULL)
    if (*curr->val == val) return *curr;
  else
    *curr = *curr->next;
}
```

Operator precedence

```
node *find(node **curr, val) {
  while (*curr != NULL)
    if (*curr->val == val) return *curr;
  else
    *curr = *curr->next;
}
```

The arrow operator `->` and the dot operator `.` bind more tightly than dereference `*`, parenthesis would solve the problem.

Control-flow

```
int x,y;
for (x=0; x<xlen; x++)
  for (y=0; y<ylen; y++);
    pix[y*xlen + x] = x*y;
```

Control-flow

```
int x,y;
for (x=0; x<xlen; x++)
  for (y=0; y<ylen; y++);
    pix[y*xlen + x] = x*y;
```

A rogue ; terminates the statement in the second loop and the assignment will only be executed once. Only the (out-of-bounds) write `pix[ylen*xlen+xlen] = xlen*ylen` will be executed. Such errors may result in partial initialization, allowing an adversary to leak information.

Control-flow

```
if (isbad(cert))
  goto fail;
if (invalid(cert))
  goto fail;
  goto fail;
```

Control-flow

```
if (isbad(cert))
  goto fail;
if (invalid(cert))
  goto fail;
  goto fail;
```

A double goto executes in any case (it is no longer scoped by the if) and always errors out. This was the famous goto fail bug in Apple's SSL implementation.

Control-flow

```
if (isbad(cert))
  goto fail;
if (invalid(cert))
  goto fail;
  goto fail;
```

A double goto executes in any case (it is no longer scoped by the if) and always errors out. This was the famous goto fail bug in Apple's SSL implementation.
A hat tip to all Apple fans.

Use-after-free

```
Node *ptr = (Node*)malloc(sizeof(Node));
ptr->val = getval();
free(ptr);
search(ptr);
```

Use-after-free

```
Node *ptr = (Node*)malloc(sizeof(Node));
ptr->val = getval();
free(ptr);
search(ptr);
```

A memory object is used after it has been deallocated.

Type confusion arises through illegal downcasts

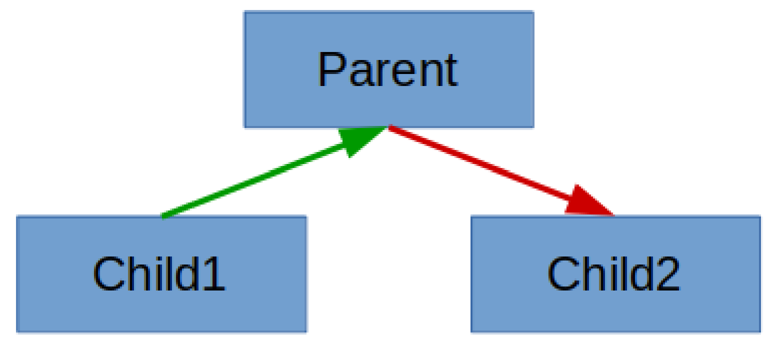

Figure 2:

```
Child1 *c = new Child1();
Parent *p = static_cast<Parent*>(c);  // OK
Child2 *d = static_cast<Child2*>(p);  // Fail!
```

Summary

- Memory safety bugs allow program state modification
  - Spatial memory safety focuses on bounds
  - Temporal memory safety focuses on validity
- Type safety ensures that objects have the correct type
- Large amounts of bug classes lead to fun vulnerabilities