CMPSC 447
Type Errors

Trent Jaeger
Systems and Internet Infrastructure Security (SIIS) Lab
Computer Science and Engineering Department
Pennsylvania State University
Type Errors

• Errors that permit access to memory according to a multiple, incompatible formats
  ‣ These are called type errors
  ‣ Access outside the expected “type”

• Most of these errors are permitted by simple programming flaws
  ‣ Of the sort that you are not taught to avoid
  ‣ Let’s see how such errors can be avoided

• Some of the changes are rather simple
Temporal Errors

- A few of the exploits that we have discussed are the result of temporal errors
Type Confusion

- Many effective attacks exploit data of another type

```c
struct A {
    struct C *c;
    char buffer[40];
};

struct B {
    int B1;
    int B2;
    char info[32];
};
```
Type Confusion

- Adversary can abuse ambiguity to control writes

```c
struct A {
    struct C *c;
    char buffer[40];
};

struct B {
    int B1;
    int B2;
    char info[32];
};

x = (struct A *)malloc(sizeof(struct A));
y = (struct B *)x;
y->B1 = adversary-controlled-value;
x->c->field = adversary-controlled-value-also;
```
Type Confusion

• Adversary can abuse ambiguity to control writes

```c
struct A {
    struct C *c;
    char buffer[40];
};

struct B {
    int B1;
    int B2;
    char info[32];
};

x = (struct A *)malloc(sizeof(struct A));
y = (struct B *)x;
y->B1 = adversary-controlled-value;
x->c->field = adversary-controlled-value-also;
```

• Arbitrary Write Primitive!
  ‣ Adversary controls the value to write and the location of the write
  ‣ Allow adversary to write an arbitrary value to an arbitrary location
What Is Going Wrong?

- We have objects (memory regions) and references (pointers)
  - What goes wrong in type errors?
How Do Type Casts Work?

- We have **objects** (memory regions) and **references** (pointers)
  - What goes wrong in temporal errors?
- A pointer may **reference a memory region using two different types** (i.e., memory formats)
- Normal lifecycle between a pointer and object
  - t1 *p, t2 *q;
  - p = (t1 *) malloc(sizeof(t1)); // allocate object and define p
  - p->field = value; // use pointer for t1
  - q = (t2 *)p;
  - q->X = value2; // use pointer for t2
How Do Type Casts Work?

- We have **objects** (memory regions) and **references** (pointers)
  - What goes wrong in temporal errors?
- A pointer may **reference a memory region using two different types** (i.e., memory formats)
- Normal lifecycle between a pointer and object
  - `t1 *p, t2 *q;` // declare pointers
  - `p = (t1 *) malloc(sizeof(t1));` // allocate object and define `p`
  - `p->field = value;` // use pointer for `t1`
  - `q = (t2 *)p;` // type cast and define `q`
  - `q->X = value2;` // use pointer for `t2`
How Do Type Casts Work?

- We have **objects** (memory regions) and **references** (pointers)
  - What goes wrong in temporal errors?
- A pointer may **reference a memory region using two different types** (i.e., memory formats)
- Normal lifecycle between a pointer and object
  - `<t1 *>p, ` `<t2 *>q;` // declare pointers
  - `p = (t1 *) malloc(sizeof(t1));` // allocate object and define `p`
  - `p->field = value;` // use pointer for `t1`
  - `q = (t2 *)p;` // **type cast** and define `q`
  - `q->X = value2;` // use pointer for `t2`
What Can Go Wrong?

- A pointer may reference a memory region using two different types (i.e., memory formats)
- Normal lifecycle between a pointer and object
  - `t1 *p, t2 *q;` // declare pointers
  - `p = (t1 *) malloc(sizeof(t1));` // allocate object and define `p`
  - `p->field = value;` // use pointer for `t1`
  - `q = (t2 *)p;` // type cast and define `q`
  - `q->X = value2;` // use pointer for `t2`
- Semantics of "`p->field`" may be different than "`q->X`"
  - Even if these reference the same memory location
What Can Go Wrong?

- **Downcasts** – Cast to a larger type; causes overflow
  - `t1 *p, t2 *q;` // declare pointers
  - `p = (t1 *) malloc(sizeof (t1));` // allocate t1 object, define p
  - `p->field = value;` // suppose this is an int field
  - `q = (t2 *)p;` // downcast, t2 is a larger type
  - `q->extra = value2;` // overflow memory of object

- **E.g., t2 is a child type of t1**
  - So, the size of type t2 is greater than the size of type t1
  - “extra” field is added to the type t1 to create type t2
What Can Go Wrong?

- **Downcasts** – Cast to a larger type; causes overflow
  - `t1 *p, t2 *q;`  // declare pointers
  - `p = (t1 *) malloc(sizeof (t1));`  // allocate t1 object, define p
  - `p->field = value;`  // suppose this is an int field
  - `q = (t2 *)p;`  // down cast, t2 is a larger type
  - `q->extra = value2;`  // overflow memory of object

- By downcasting to the larger type `t2` with the “extra” field, gives the adversary the ability to read/write beyond the memory region allocated
  - Memory region is the “sizeof(t1)” in size
Exploiting Type Errors

• Type \( t_2 \) is a child type (downcast) of type \( t_1 \)

```
<table>
<thead>
<tr>
<th></th>
<th>Int F1</th>
<th>Int F2</th>
<th>Int F3</th>
</tr>
</thead>
</table>
```

• Allocate object of type \( t_1 \) and assign “p” to reference
Exploiting Type Errors

- Type t2 is a child type (downcast) of type t1

<table>
<thead>
<tr>
<th>“p”</th>
<th>Int F1</th>
<th>Int F2</th>
<th>Int F3</th>
</tr>
</thead>
</table>

| “q” | Int F1 | Int F2 | Int F3 | Int extra |

- Assign “q” of type t2 to the memory location of “p”
  - But, ”q” of type t2 thinks it is referencing a larger region
Exploiting Type Errors

- Type t2 is a child type (downcast) of type t1

<table>
<thead>
<tr>
<th>“p”</th>
<th>Int F1</th>
<th>Int F2</th>
<th>Int F3</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>“q”</th>
<th>Int F1</th>
<th>Int F2</th>
<th>Int F3</th>
<th>Int extra</th>
</tr>
</thead>
</table>

- What will happen when the program accesses “q→extra”?
What Can Go Wrong?

- **Type confusion** – use data to craft a pointer
  - `t1 *p, t2 *q;`  // declare pointers
  - `p = (t1 *) malloc(sizeof(t1));`  // allocate object and define p
  - `p->field = value;`  // suppose “field” is an int field
  - `q = (t2 *)p;`  // type cast and define q
  - `q->X->target = value2;`  // suppose “X” is a pointer field

- “p->field” is a data field, so may store adversary data
- But, “q->X” is a pointer field
  - Should we allow adversaries to define pointer values?
What Can Go Wrong?

- **Type confusion** – use an integer to craft a pointer
  - \( t1 \ast p, t2 \ast q; \) // declare pointers
  - \( p = (t1 \ast) \text{malloc} \text{(sizeof}(t1)); \) // allocate object and define \( p \)
  - \( p \rightarrow \text{field} = \text{value}; \) // suppose “field” is an int field
  - \( q = (t2 \ast)p; \) // type cast and define \( q \)
  - \( q \rightarrow X \rightarrow \text{target} = \text{value2}; \) // suppose “\( X \)” is a pointer field

- The write to “field” of type “\( p \)” gives the adversary the ability to choose a memory location for the write to “\( X \)” – if at the same offset as “field”
  - To modify an adversary-chosen memory location
  - Relative to the field “target”
Exploiting Type Errors

- Type $t_2$ is unrelated to type $t_1$

| “p” | Int F1 | Int Field | Int F3 |

- Allocate object of type $t_1$ and assign “p” to reference
Exploiting Type Errors

- Type t2 is unrelated to type t1

```
<table>
<thead>
<tr>
<th></th>
<th>Int</th>
<th>Int</th>
<th>Int</th>
</tr>
</thead>
<tbody>
<tr>
<td>“p”</td>
<td>F1</td>
<td>Field</td>
<td>F3</td>
</tr>
<tr>
<td>“q”</td>
<td>Int</td>
<td>Ptr</td>
<td>Int</td>
</tr>
</tbody>
</table>
```

- The offset of “Field” from “p” of type t1 and “X” from “q” of type t2 are the same
Exploiting Type Errors

- Type $t_2$ is unrelated to type $t_1$

<table>
<thead>
<tr>
<th>“p”</th>
<th>Int F1</th>
<th>Int Field</th>
<th>Int F3</th>
</tr>
</thead>
<tbody>
<tr>
<td>“q”</td>
<td>Int F1</td>
<td>Ptr X</td>
<td>Int F3</td>
</tr>
</tbody>
</table>

- Assign an adversary-controlled value at “$p \rightarrow \text{Field}$”
Exploiting Type Errors

- Type t2 is unrelated to type t1

<table>
<thead>
<tr>
<th>“p”</th>
<th>Int F1</th>
<th>Int Field</th>
<th>Int F3</th>
</tr>
</thead>
</table>

| “q” | Int F1 | Ptr X | Int F3 |

- But, program accesses “q→X” as a pointer
  - What can an adversary do?
Exploiting Type Errors

- Type t2 is unrelated to type t1

<table>
<thead>
<tr>
<th>“p”</th>
<th>Int F1</th>
<th>Int Field</th>
<th>Int F3</th>
</tr>
</thead>
</table>

| “q”  | Int F1 | Ptr X     | Int F3 |

- But, program accesses “q→X” as a pointer
  - Adversary chose the address stored at ”q→X”
  - Thus, the adversary can choose the memory to access
Exploiting Type Errors

- Project 2
  - What object can be accessed by pointers of multiple types?
  - Is there a flaw that allows you to create an object of one type ...
  - And access that object with a pointer of a different type?
  - Can you find a target that you can modify using the mismatched type’s pointer?
  - What do you want to exploit if you can modify the target?
  - Craft the payload to cause a modification that implements the desired exploit
Exploiting Type+Temporal Errors

- With temporal errors

- Can also exploit type confusion using temporal errors, such as *use-after-free*
  - Obj B of type B is deallocated, but has a stale pointer “b”
  - Obj D of a different type D is allocated in that free slot
  - Then, a use-after-free flaw can use “b” of type B to access Obj D of a different type D
• Yet, another issue (probably the last one) to consider
  ‣ NOTE: Very different from buffer overflows
• Key question
  ‣ What is an integer?
  ‣ In a computer system?
Integer Overflows

• Yet, another issue (probably the last one) to consider
  ‣ NOTE: Very different from buffer overflows

• Key question
  ‣ What is an integer?
  ‣ In a computer system?

• There are several different computer representations for integers
  ‣ Size – number of bytes used to represent
  ‣ Signedness – range of values integers can take
Integer Overflows

- Suppose we have an 8-bit integer type
  - How many values can it represent?
  - What range of values can it represent?
Integer Overflows

- Suppose we have an 8-bit, signed integer type
  - How many values can it represent?
    - \(2^8 = 256\)
  - What range of values can it represent?
    - Depends on whether it is “signed” or not
  - What are the range of values if “unsigned”?
    - 0 to 255
  - What are the range of values if “signed”?
    - -128 to 127
Integer Overflows

• Can you attack this?

```c
int x;
char *buf = ( char *)malloc( 50 );
x = adversary-controlled-value;
If ( x < 50 ) {
    snprintf( buf, x, "%s", adversary-controlled-input );
}
```
Integer Overflows

- Can you attack this?
  - Unfortunately, we can – snprintf casts to unsigned
  - Negative value becomes a large positive value

```c
int x;
char *buf = (char *)malloc(50);
x = adversary-controlled-value;    // negative value
If ( x < 50 ) {                    // passes this condition
    sprintf( buf, x, "%s",       // second arg ‘x’ to snprintf is unsigned
        adversary-controlled-input );  // too long input
}
Integer Overflows

- Can you attack this?
  - Unfortunately, we can – have to compute “size” correctly

```c
int x;
char *buf = ( char * )malloc( 50 );
x = adversary-controlled-value;    // negative value
If ( x < 50 ) {                    // passes this condition
    snprintf( buf, x, “%s”,      // ‘x’ becomes a large positive value - overflow
               adversary-controlled-input );  // too long input
}
Fundamental Problem?

- What is the **fundamental problem** that causes type errors?
Fundamental Problem?

- What is the fundamental problem that causes type errors?
  - Type casting – create pointer of different type
  - Temporal errors – change type of memory region

- These enable the same memory region to be referenced as multiple types
  - Enabling exploitable type errors

- Resulting exploitable flaws
  - Misinterpret the size of the region (downcast)
  - Data misinterpreted as a pointer (type confusion)
  - Data values misinterpreted (integer overflow)
Safety from Type Errors

- **Type safety**
  - Memory region is only referenced by pointers of one type
  - Corresponding to the type of the memory region allocation

- **Memory safety** (for regions of multiple types)
  - Memory region may be referenced by pointers of more than one type
  - Semantics of all references correspond to allocation and consistent use of the memory region
  - Think about “question” types in the project
Obvious Solution in C

• So, do you see an “obvious” solution to prevent exploitable type errors?
Obvious Solution in C

- So, do you see an “obvious” solution to prevent exploitable temporal errors?
  - No type casts? May be hard to ensure that
Obvious Solution in C

• So, do you see an “obvious” solution to prevent exploitable temporal errors?
  ‣ What can we do to prevent the misinterpretation of data even if we allow type casts?
Safe Casts

- Only allow "upcasts" for type casts
  - A “downcast” from a parent data type to a child data type
    - Adds more fields – may allow overflow
  - An “upcast” from a child data type to a parent data type
    - Reduces fields – no overflow possible, fields are same type
Other Ideas

- Can you think of any other ways to prevent type error exploits?
  - May be a little crazy
Alternatives

- **Hypothesis**: Validate type consistency on casts
  - At runtime – but can be expensive (>100%)
  - Maybe type casts are not super-common in your program
  - Prove some type casts are safe statically?

- **Hypothesis**: Use type-specific allocation
  - Only helps for temporal errors
  - Unless do validity checks also
Take Away

- Flexible type casting in C permits type errors
  - So, type errors have become common, especially now that defenses for spatial errors have improved

- Exploiting type errors involves exploiting a reference to a memory region interpreted in multiple ways (using multiple types)
  - Set data value, but use as a pointer

- Preventing type errors is not so easy (except upcasts)
  - And, a bit more expensive than people will accept yet