CSE 409/509 - Spring '09: System Security - Lecture Notes

Date: 03/06/2009
Notes by: Kaushik Chatterjee

Topic: "CCured"

Basic outline of Program Transformation:

Desired Properties of Program Transformation:

- Code generated by the *Rewriter* should be bug free and continue working.
- Should not affect performance (or have as little impact on performance as possible).
- The O/P should be free of bugs.
- Least manual effort should be required (require less or no change to the source code).
- Transformed code should be compatible with the Un-Transformed code.
- Require little or no annotation.

CCured:

C language divided into 3 subsets.

- SAFE
- SEQ
- DYNAMIC

Pointer operations in C

- Arithmetic *(Unsafe)*
- Dereference
- Cast between pointer types *(Unsafe)*
- Can point anywhere
- Cast pointer to int/ implicit conversion
- Cast int to pointer/ implicit conversion *(Unsafe)*
- Assignment/ Parameter passing
- &
- Allocation/ Free
- NULL
- void *
- Pointer to a function.

Temporal Memory Bugs:

- Use of pointers after they have been freed:

```
p = malloc(10);
free(p);
*p = 0;
```

- Pointer Escaping:

```
int * foo(void) {
 int x;
 return &x;
 }

void bar(void) {
 int * p = foo();
*p=0;
}
```

However, CCured deals mainly with Spatial Memory Bugs, and thus we leave the discussion of Temporal Memory bugs for the time being.

We deal with 3 different subset of the C language:

SAFE:

Only the following pointer operations (which are safe), are allowed:

- Dereference
- Cast pointer to int/ implicit conversion
- Assignment/ Parameter passing
- &
- Allocation/ Free
- NULL
- Pointer to a function.

How to deal with function 'memcpy', if incorrect size is passed

```
Using: void * memcpy(void * dst, void * src, int sz)
int * p;
int * q;
memcpy(p, q, sizeof(* q));
```

This however consist implicit conversion between pointer types (int * to void *):

void * memcpy((void *) p, (void *) q, sizeof(* q))

Hence, this is not allowed as per the rules, and we are safe.

References in C++

- Null cannot be passed
- No pointer arithmetic operations allowed

```
int foo(int & x){
x=5;
 }
```

Thus for SAFE, Runtime Checks:

- None
- Maybe – Check for NULL

SEQ:

SEQ = SAFE + Pointer Arithmetic + integer to pointer casting.

- Needs “bounds checking”. Thus we need to keep track of bounds.
- No type inconsistent aliasing. (ie. compiler knows statically the type of data pointed to by every pointer)

```
struct S{
int x;
};

struct P{
 int *p;
 };

struct S s;
struct P *p;

 s.x=<any value>;
 p=(struc P*)&s;
 *(p->p)=0;
```

Thus, we need pointers that cannot be *dereferenced*. Casts from integer to pointer must yield an *undereferencable pointer*.

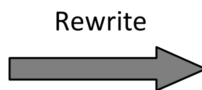
Pointer Representation in SEQ:

Pointer **p** is represented as a triplet **<l, h, p>**, where:

- l: Lower Bound
- h: Higher Bound
- p: actual pointer value

```
*p = 0;
```

Rewrite



```
If (p<l || p>h)
    abort();
*p = 0;
```

For undereferencable pointers p := <0, 0, p> (ie: l = h = 0).

Conversion of pointers from SEQ to SAFE:

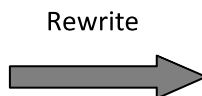
```
void foo(int *p) {
 *p=0;
 *p=1;
*p=2;
 }                          } SAFE

void bar(void) {
int A[5];
int *p;
p=A;
p=p+3;
foo(p);
 }                          } SEQ
```

Here p := <&A, &A[4], p>

Therefore foo(p) will be replaced by

```
foo(p);
```

Rewrite



```
If (p<&A || p>&A[4])
      abort();
foo(p);
```

Ie. we need to perform a ‘Bounds Checking’.