CSE409, Rob Johnson, http://www.cs.stonybrook.edu/~rob/teaching/cse409-fa11/
Alin Tomescu, November 14th, 2011

Static and dynamic compiler-based defenses

Overview of defenses

We covered:

- Model checking
- CQual, type qualifiers (tainted and untainted)
- CCured
- Deputy, library compatibility

	False positives/negatives	**Library compatibility**	**Code compatibility**	**Static vs. Dynamic**	**Speed slowdown**	**Space overhead**
Deputy	0	Yes	No, you have to add annotations.	It's dynamic, but also does some static optimizations	15%	~0
Jones & Kelly	0	Yes	Yes, with some minor changes maybe.	Dynamic	1000%	High
CQual	20%	No	Pretty good, you only need to add the $tainted qualifiers.	Static	0%	0%
Model checking	High	Yes	Yes	Static	0%	0%
Run-time taint tracking						

Jones and Kelly bounds checker

A dynamic system, it has a **zero false-positive rate**, but the overhead is around 1000%.

Jones and Kelly is famous for its **simplicity**. Their system keeps track of all your bounds for you and it maintains the invariant that all pointers are always in bound.

The tracking is done using **interval trees.**

- I have a bunch of **non-overlapping intervals** (pointer bounds)
- I have a pointer
- I want to quickly find the bounds for the pointer

An **interval tree** allows you to do the following operations in $\log n$:

- `insert(a, b)`, inserts the interval (a, b) into the tree
- `(a,b) = lookup(c)`, where c is a number, checks if there exists an interval (a, b) containing c and returns it
- `remove(a, b)`, removes the interval (a, b) from the tree

In practice, **splay trees** are used because of their **self-balancing** and **caching** properties.

If I do $p = q + 5$, how do I maintain p's bounds?

- I can do a $(a, b) = lookup(q)$ before the assignment,
- And check after the assignment: `if(q < a || q ≥ b) abort();`

If I do, `p = malloc(10)`, then I should `insert(p, p + 10);`

When you do `free(p)`, then you should remove the interval for p in the tree

What about doing `*p`? `p` is guaranteed to be in bounds, so `p` does not need to be checked

What if `sizeof(*p) = 16`. Then if p was in bounds `[start, start + 16)`, `*p` could still be accessing invalid memory when p is in bounds. For instance if `p = start + 5`, then `*p` will be accessing memory locations from `[start + 5, start + 5 + 16)`, which is past the bounds.

Therefore, we need to **take the size of the pointee in consideration** when performing bounds checking.

```
(a, b) = lookup(p);
if(p + sizeof(*p)) > b)
      abort();
else
      *p;
```

Slight issue: Library compatibility. When libraries allocate or free memory for you, you can't add the bounds to the interval tree anymore.

Bigger issue: Jones & Kelly will never let pointers go out of bounds, and sometimes they need to.

```
for(int i = 0; i < n; i++) *p++ = 0;
```

p will be out of bounds at the end of the loop, so this would be an error. Jones & Kelly always **allocate an extra byte** to solve this little incompatibility.

Pointers that go out of bounds, are typically pointers on the stack, so another approach is to let a pointer p wander around and only check its bounds when it's actually being dereferenced.

CQual

A lot of bugs depend on **data-flow in your code**, such as format-string bugs, SQL injection bugs or XSS bugs.

CQual is a system for **tracking the flow of data** in your program at **compile-time**, through a formalism called **type qualifiers.**

```
void foo($tainted char * username)
{
      char * s = username;
      char * t = s;

      printf(t);
}
```

`printf` is actually declared like `int printf($untainted char * fmt, ...);`

- Mark all the functions that `get`/return tainted data, and the functions that take/return untainted data.
- Compiler makes a graph for $\$tainted\ char \rightarrow * username \rightarrow * s \rightarrow * t \rightarrow * fmt$ (the argument of `printf`)
- If a **path from tainted data to untainted data** is found in your program, then that will be marked as an error.

CSE409, Rob Johnson, http://www.cs.stonybrook.edu/~rob/teaching/cse409-fa11/
Alin Tomescu, November 14th, 2011

Run-time taint tracking

Run-time taint tracking was pioneered at Stony Brook by professor Sekar. The system **tracks taint at the byte level**.

When your program starts, a **tainted tag map** for each byte is allocated, which indicates whether byte i is tainted.

```
tagmap = malloc(2^32/8);
memset(tagmap, 0, 2^32/8);
```

This map has to **track all the bytes in a program's virtual address** space so it needs 2^32 entries on a 32-bit system. Therefore, **compression techniques** are used to ensure its size is reasonable

When you do a read, you have to **set the bytes you read into as tainted**.

```
read(fd, buf, len);
for(int i = 0; i < len; i++)
      tagmap[buf + i] = 1;
```

If you **assign a buffer b to a buffer a,** you have to also assign b's tag-map to a's tag-map.

```
a = b;
for(int i = 0; i < sizeof(a); i++)
      tagmap[&a + i] = tagmap[&b + i];
```

When you come to a dangerous operation, such as `printf`, you can check its arguments. The **goal is to be as tolerant as possible**, and only abort the program when the tainted bytes are evil. We do not want to abort the program just because the bytes are tainted. Obviously, any user input will be tainted.

```
//    Check that printf's arguments are not evil, by ensuring its
//    tainted bytes are not equal to %.
printf_check(foobar);
printf(foobar);

void printf_check(char * s)
{
      while(*s)
      {
            if(*s == '%' && tagmap[s])
                  abort();
      }
}
```

An attacker can still overflow a return address, so **return addresses should be checked as well**.

Note that the overflown bytes will be tainted as the attacker overflows the buffer, so the following simple check can easily tell if the return address ret was overflown.

```
void return_check(void ** r)
{
      for(int i = 0; i < sizeof(*r); i++)
            if(tagmap[r+i] == 1)
                  abort();
}
```