# Trusted Case Study – Apache Chukwa

The trusted type system is extremely simple to understand, but I found it was not simple to use, at least for this particular code base. I found myself casting Untrusted Strings to Trusted Strings very frequently, suppressing warnings on common false positives, and writing what felt like extra annotations.

## Reasons for Casting

1. Unannotated Libraries

Some library functions should always return a Trusted value. For example, executing a query creates a ResultsSet, which contains all the rows and columns from the database that match the query. Any information that comes from a ResultSet should be Trusted because it comes from the database, a Trusted source. However, without an annotated version of ResultSet, the information must be cast to Trusted.

@Trusted String result = (@Trusted String) resultSet.getString();

2. Manipulating Trusted Strings

Often, new Trusted Strings are built from pieces of other Trusted Strings. In one case, the developers used templates for queries and used regular expressions to create real queries out of these templates, replacing placeholders with things like a table name. If the table name is Trusted, then the overall query would be trusted as well and needed to be cast as such. Methods such as String.replace(), String.split(), etc. needed to have their return values case to Trusted on a case-by-case basis.

I also saw StringBuffers used to create Strings that were logically Trusted, such as to insert a partition number in the middle of a Trusted string. While I could verify these cases logically, the resulting string still needed to be cast.

@Trusted String str = (@Trusted String) buffer.toString();

3. Primitives concatenated into Trusted Strings

In a few cases, queries were built up using concatenation and included a partition number. These numbers were a result of a calculation, and the Trusted type system does not understand numeric manipulation. So, it did not work to just annotate these numbers as Trusted at their declaration; I would still have to cast them at every

mathematical operation. The easiest solution was to cast them to Trusted once at the end as they were being used in the query.

```
@Trusted String query = “foo ” + (@Trusted int) partition + “ bar”;
```

## Other Common Issues

1. Trusted String is initially null

In several places, a certain String could be either null or a few specific String literals. If it was non-null, it was one of those Trusted String literals and used to create a new Trusted String. Or, a string was simply declared as null, and then later in the method assigned to a literal using a series of if-else statements. These initially-null Strings are clearly Trusted, but assigning a Trusted String as null is a type error. Since they would never be used when null, I suppressed the error.

```
@SuppressWarnings(“trusted”)
@Trusted String str = null;
…
if(someCondition)
    str = “foo”;
else
    str = “bar”;
…
@Trusted String query = “someQuery” +str;
```

2. Copying a Trusted String

A copy of a Trusted String must also be a Trusted String, but the checker does not understand this. It was slightly irritating to keep getting the following errors, though they were trivial to fix.

```
this.query_state = new String("read");
                   ^
 found   : @Untrusted String
 required: @Trusted String
public void setJobName(@Trusted String s) {
   this.jobname = new String(s);
                  ^
 found   : @Untrusted String
 required: @Trusted String
```

## Did I find any errors?

I found a few slightly questionable things, but nothing concrete.

In two places, a Trusted String is taken directly out of an HTTP Request object without verification. That might be an issue because an attacker could potentially build a request with anything in it, but neither place was ever used in the codebase. It might be dead code.

In another case, a list of queries is read in from a configuration file and then each is executed. In a README which explains the configuration files, it says this file should never be modified. The permissions on this particular file are "-rw-r--r--", so it is read-only for everyone except the owner. I'm not sure if this is a security concern or not. The owner of the file could insert malicious queries, but then he would just be hacking himself.

## What might this look like as a game?

Given the number of casts that I had to use, one possibility is that players need to use a lot of buzzsaws. Perhaps they will choose to put buzzsaws as close to the problem as possible. Or, alternatively, they might choose to make more pipes narrow than necessary, and stick buzzsaws at random. It might be very easy for players to get lost; this happened to me as an annotator. A small logic mistake made me think something needed to be Trusted when it didn't, and this led to a chain-reaction of variables that would need to be Trusted. Once I realized that I had an argument to the main method flowing into a Trusted variable, I knew something was wrong and backtracked to find my mistake. I expect players may experience a similar need to backtrack or rethink their higher level strategy, but I'm not sure whether the abstraction of the game would help or hinder them with this.