2020 CTF dedup talk: speaker's notes

Nick Alcock

August 25, 2020

Contents

1 CTF basics (offer to skip to save time)

In particular, this covers the parts that caused pain in the hope that others doing the same thing can do what I ended up doing, and avoid the pain.

1.1 encoding of C types in the type section

1.2 deduplication of this section necessary when linking before you can look at linking anything else CTF-related

The rest of the linker only needs to get told how to map from input (CTF section, type) pairs to the corresponding output type. This makes the algorithm used to do deduplication relatively decoupled from the rest of the CTF linker, let alone the linker as a whole

2 Type deduplication

2.1 Avoiding allocation nightmares

We use hashtabs in preference to other data structures because it's very easy to evolve them without having to revise a data structure first. But we'd like to not have to malloc for every single thing we store in these hashtabs, and for all the hashtab keys, simply to avoid filling the code with allocations rather than useful work. A great many of the things we store fall into two classes: hash values (SHA-1s) and references to types in input TUs. There are relatively few hash values (one per deduplicated type), so we stash them in an atoms table and store a pointer to that. But the input types... there are many of these (in different data structures) per **input** type. We need efficiency.

We use them as hashtab keys, so the only property that matters is that a given type in a given input always has the same GID. (They are never compared except for equality.)

Simple encoding on 64-bit platforms: high 32 bit is integer giving position of input on link line (allowing for archives etc). Second 32 bit is ctf_id_t.

On 32-bit, we stuff the (input integer, ctf_id_t) pair into a hashtab and use the pointer to the key as the GID. Less efficient because we waste memory on the hashtab, but 32-bit links are likely to be smaller than 64-bit anyway (and 32-bit uses less memory in any case due to reduced alignment). Seems to be a wash in testing, memory-wise.

2.2 In brief: hashing, ambiguity detection and conflict marking, emission

(time mark: 5:00)

We do deduplication by recursive hashing: types with identical hashes are considered to be deduplicated and are emitted only once.

Three stages:

- hash every type in every TU recursively, TU by TU, mixing the hash of all types cited by any type into the hash of that type (so "const int" mixes in the hash of "int"). IDs and other internal details (which do not affect type identity, and are a part of our representation, rather than the type itself) are never mixed in. Remember mappings from GID -> hash and vice versa. The set of hash values corresponds to the deduplicated output if no types were ambiguously defined. We hash each GID at most once, caching it so that other types referring to it don't cause us to go O(n^2).

  This is tricky because we want identical types to end up with identical hashes while also connecting opaque forwards to structures to a non-opaque definition, if an unambiguous one can be found in some other TU: but an opaque forward surely isn't going to hash to the same value as its non- opaque struct! That's a puzzle for later...

  This is linear-time in the number of types in input TUs.

- types can be ambiguous: e.g. two different TUs with the same-named struct with different content. This is routine in C, and even seen in practice in C++, despite the ODR. We spot these by hunting for types with the same name (qualifying for struct/union/enum) but different hashes, and mark such types and all the types that cite them as conflicting.

  This is linear-time in the number of deduplicated types with names.

- emit everything by traversing the set of hashes, emitting types referred to by each, from leaf to root (if not already emitted), then emitting the types themselves.

  This is linear-time in the number of deduplicated types.

2.3 Avoiding getting stuck in cycles: strategies

The C type system is acyclic, because of forwards (slide 8); but CTF is not acyclic, because it cannot represent more than one instance of a given name in a given namespace. It can encode forwards, but replaces forwards with their referents if the referents are also added: so in this case we end up with 'struct foo', then 'struct bar', each with a member that points to the other. We'd like to collapse away forwards like this whenever possible, because it is more convenient for users to see a struct than a forward with no members:

because of ld -r, this means that inputs can contain cycles galore. Some of these cycles can be huge (thousands of members), so O(n^2) algorithms won't work: when I broke the cache above, objcopy suddenly took hours to link!

We treat structs and unions identically: so from now on, we call 'struct/union' simply 'struct'.

Worse yet, given cycles, the algorithm above obviously doesn't work! We chase everything we can, so that CTF users can go from foo $*$ to the definition of 'foo' and from 'bar $*$' to 'bar' – but now each of the steps above can get stuck.

2.4 What doesn't work

(time mark: 9:00)

I spent months on wrong tracks.

The obvious first approach is to try to come up with a representation of the type graph in which no information about what types cite other types is lost (and the graph structure is preserved), but where all cycles of the same types are identical in all TUs. This is tricky given that we can enter a cycle at any point and that cycles are often huge (libbfd contains cycles hundreds of types long), but it can be done... but it still doesn't work. [graph] The problem is that, thanks to opaque forwards, a cycle in one TU is not a cycle in all TUs, even if the types are the same and should be deduplicated. [example]

2.5 What worked

(time mark: 11:00)

So we can't keep the type graph completely intact: we have to break cycles somehow, at least when forwards can point into the cycle (in some other TU which we cannot see at the time we are hashing the cycle), and we have to break them in a way that is the same in a TU where the cycle is visible and in a TU where we only point into it, and we should also break them in a way that somehow makes structures and forwards equivalent without mixing all structures up.

We don't want to do too much damage to the type graph – the limit case would be declaring all types to be independent of all others and not chasing cited types at all when hashing, which means we'd consider 'const int' to be identical to 'const char $*$' (they're both a const **something** after

all). What's the least damage we can do to the type graph while eliminating cycles?

We just saw why we can't hash things differently if a type is in a cycle: we might not know it's in a cycle from inside this TU.

(graphviz examples)

The key is tagged structures/unions. Every cycle in the C type graph must contain at least one of these, and forwards must be forwards to one of these too. When we see a tagged struct/union or a forward to one while hashing types to mix them in to a parent type, hash it as a **stub**. Stubs are literally just the name of the struct/union: no content, no child types. Stubs do not hash the same as anything but other stubs. (All **actual** types get a lot more than just a type name mixed in, so this cannot collide with anything else). Now a type that cites an opaque forward is hashed identically to a type that cites its non-opaque equivalent, and we can deduplicate them together.

This is not the only place we can get stuck in cycles, but it's the trickiest to resolve. More later.

Ludicrous types: the 'struct bfd' cycle has a thousand-plus types in it, and some of those types are defined differently for different targets!

2.6 Ambiguities and the popcon

(time mark: 16:00)

What if we have two TUs that both refer to a struct with the same name, but that struct is different in both TUs? The algorithm above will consider them identical, but they really aren't. We have to fix this somehow.

Now we have a bunch of hashes, we can spot cases where a single C type name has multiple distinct hashes associated with it. This is something that cannot happen in a single TU: some type must have the same name but a different definition in different TUs, i.e. it is ambiguous. CTF cannot represent this in one dictionary, since a CTF dictionary roughly corresponds to a C file-level scope: so we note this problem by marking such types **conflicted**.

[graph of conflicted marking]

A conflicted type is emitted into a child dictionary that is specific to one TU: it can cite types in the shared dict but not types in other child dicts (but since all of those types are ambiguous types in other TUs, this is fine). However, since the shared dict has many child dicts, no shared type can cite a type that is not shared: so we have to walk **backwards** up the type graph, from types to the types that cite them, and mark them all conflicted too.

This is a waste of space which will be resolved in format v4... but also it means we can get stuck in a cycle again! We resolve this in exactly the same way: we break the graph of types-that-cite-this-type at tagged structures and stop marking things as conflicted when we hit a tagged structure. This also means we don't have to move entire cycles into child dicts.

We can reduce the amount of conflicted types we generate by finding the most- commonly-cited ambiguous type and **not** marking it conflicted, so it can stay shared: so only the rarer types cause space bloat. (This is not yet implemented for structures or unions, where we have to duplicate everything: v4 will fix this.)

(There is one other thing this pass does. If we get a hash collision of a named type and an unnamed type, or of two named types, ambiguity resolution will spot it and fix the problem by considering the two types ambiguous. If we get a hash collision of two **unnamed** types, we don't spot it and we get a corrupted type graph, but this is massively unlikely and I really don't care about it: if it ever happens we can move to something other than SHA-1 without affecting users at all. I frankly doubt anyone will notice: our existing CTF emitter does nothing about ambiguous types at all and just outright emits corrupted type graphs and nobody has noticed in eight years other than me.)

2.7 Consequences for emission

(time mark: 19:00)

Actually emitting types is easy once we have all of this. We walk over the hashes of all types we know about, leaf to root: we emit unconflicted types into the shared dict and conflicted types into N dictionaries (one per input TU in which that type appears). It is impossible for an unconflicted type to cite a conflicted type at this point: if it happens we assert and die because it must be a bug (and did we ever hit that assertion a lot during debugging).

[graph]

There are two places where we have to do special things. Firstly, because we stop marking things conflicted at tagged structures, we **can** have an unconflicted type cite a conflicted tagged structure. We spot this and emit a **synthetic** opaque forward (not corresponding to any forward in the input) in the shared dict: this will need user intervention to climb past to get to the real structure, but that's unavoidable because there's more than one structure it might be and we have no idea which structure it is! (The input C code literally doesn't know: all it's got is an opaque forward, but there are

several distinct structures with that name in different TUs. It takes semantic knowledge to resolve this, and we don't have it, but the user does.)

It's not technically necessary to avoid cycles, but it does simplify things a bit (and reduce the recursion depth) to emit structure members late. When we emit a structure we don't emit its members: we only note that its members need to be emitted. Then we have another pass that whips through all the structures we've noted we need to emit members for, and emit all of them.

3 Extension to other languages

(time mark: 21:00)

I... haven't thought about this much. For ObjC and possibly C++ the algorithm should be almost unchanged: as with C the only cycles possible are with tagged structs/unions/classes, I believe? (Though C++ is going to need the most enormous extension to CTF: I will need input from people who used C++ more recently to know what bits of the type graph it even makes **sense** to encode for runtime lookup.)

4 Other stuff

There's not very much left. We construct and export a mapping so the rest of the CTF linker can tell what input type maps to what output type (inefficiently – this is something that we can and will improve when we rip out the old non- deduplicating linker so we don't need to carry around a mapping that works for it too).

We link other sections much more simplemindedly: very little deduplication is needed, because the type section dedup has done nearly everything for us. The only complexity is in the function and object info sections, which in the original Solaris model are big arrays with one entry per suitable symtab entry (though a lot are elided), giving the types associated with that symbol. This is obviously not acceptable for child dicts containing conflicted types: so instead there we emit the arrays in arbitrary order, and have index sections that map symbol names to array entries (this is also the same form the compiler gives it to the linker in). We use this form if the extra cost of storing the index sections is less than the cost of storing null entries in the arrays corresponding to all the symtab entries that don't use any types in this child dict at all.

5 Acknowledgements and prior art

- libabigail, for proving that hashing types isn't a disaster
- the BTF deduplicator, which uses a wildly different algorithm
- the previous DWARF -> CTF deduplicator I wrote, which is an object lesson in how not to do it
- Solaris's deduplicator, which is different again, and very Solaris- kernel-specific