The GNOME™ Conference

GUADEC

We Cannot Write Secure Applications

Michael Catanzaro (mcatanzaro@gnome.org)

Presentation Outline

- Memory Safety
- Supply Chain Security
- Sandboxing

Memory Safety

Memory Safety 101

- Unsafe languages: C, C++, Vala
- Safe languages: JavaScript, Python, Rust
- In unsafe languages, common errors allow attackers to control users' computers
- Attacking is hard; I'm a defender and don't know much about attacking
- Defenders simply assume every memory safety error is a security vulnerability
  - Write errors: code execution exploits
  - Read errors: leak sensitive data or facilitate exploitation of write errors

Memory Safety 101: Unrealistic Example Errors

- Buffer overflow

```
int data[42];
data[42] = 0;
```

- Use after free

```
int *data = malloc ();
free (data);
data[0] = 0;
```

Memory Safety 101

- High-risk applications: Epiphany, Totem/Showtime, Evince/Papers
- Our errors have serious consequences for users

Common Mistake #1: Failure to Disconnect Signal Handler

```
static void
some_signal_cb (gpointer user_data)
{
  A *self = user_data;
  a_do_something (self);
}

static void
some_method_of_a (A *self)
{
  B *b = get_b_from_somewhere ();
  g_signal_connect (b, "some-signal", (GCallback)some_signal_cb, a);
}
```

Solution to Mistake #1: g_signal_connect_object() or g_clear_signal_handler()

```
static void
some_signal_cb (gpointer user_data)
{
  A *self = user_data;
  a_do_something (self);
}
static void
some_method_of_a (A *self)
{
  B *b = get_b_from_somewhere ();
  g_signal_connect_object (b, "some-signal",
                           (GCallback)some_signal_cb, a, 0);
}
```

Common Mistake #2: Misuse of GSource Handler ID

```
static gboolean
my_timeout_cb (gpointer user_data)
{
  A *self = user_data;
  a_do_something (self);
  return G_SOURCE_REMOVE;
}

static void
some_method_of_a (A *self)
{
  g_timeout_add (42, (GSourceFunc)my_timeout_cb, a);
}
```

Non-preferred Solution to Mistake #2: Ref the User Data

```
static gboolean
my_timeout_cb (gpointer user_data)
{
  A *self = user_data;
  a_do_something (self);
  g_object_unref (a);
  return G_SOURCE_REMOVE;
}
static void
some_method_of_a (A *self)
{
  g_timeout_add (42, (GSourceFunc)my_timeout_cb, g_object_ref (a));
}
```

Better Solution to Mistake #2: g_clear_handle_id()

```
static void
some_method_of_a (A *self)
{
  a->my_timeout_id = g_timeout_add (42, (GSourceFunc)my_timeout_cb, a);
}

static void
a_dispose (GObject *object)
{
  A *a = (A *)object;
  g_clear_handle_id (&a->my_timeout_id, g_source_remove);
  G_OBJECT_CLASS (a_parent_class)->dispose (object);
}
```

Common Mistake #3: Failure to Cancel Asynchronous Function

```
static void
some_method_of_a (A *self)
{
  B *b = get_b_from_somewhere ();
  b_do_something_async (b, NULL /* cancellable */, a);
}
```

Non-preferred Solution to Mistake #3: Ref the User Data

```
static void
some_method_of_a (A *self)
{
  B *b = get_b_from_somewhere ();
  b_do_something_async (b, NULL, g_object_ref (a)); // unref in callback
}
```

Better Solution to Mistake #3: GCancellable

```
static void
some_method_of_a (A *self)
{
  B *b = get_b_from_somewhere ();
  b_do_something_async (b, a->cancellable, a);
}
static void
a_dispose (GObject *object)
{
  A *a = (A *)object;
  g_cancellable_cancel (a->cancellable);
  g_clear_object (&a->cancellable);
  G_OBJECT_CLASS (a_parent_class)->dispose (object);
}
```

Better Solution to Mistake #3: GCancellable

```
static void
something_finished_cb (GObject      *source_object,
                       GAsyncResult *result,
                       gpointer      user_data)
{
  B *b = (B *)source_object;
  A *self = user_data;
  g_autoptr (GError) error = NULL;
  if (!b_do_something_finish (b, result, &error)) {
    if (!g_error_matches (error, G_IO_ERROR, G_IO_ERROR_CANCELLED))
      g_warning ("Failed to do something: %s", error->message);
    return;
  }
  a_do_something_else (self);
}
```

Common Mistake #4: Incorrect Use of GMainContext

- Study the main context tutorial thoroughly, especially if developing a library
- https://developer.gnome.org/documentation/tutorials/main-contexts.html

Common Mistake #5: Failure to Disconnect Weak Pointer

```
static void
a_start_watching_b (A *self,
                    B *b)
{
  self->b = b; // When b is destroyed, self->b will be set to NULL.
  g_object_add_weak_pointer (b, &self->b);
}
static void
a_do_something_with_b (Foo *self)
{
  if (self->b)
    // Do something safely here, knowing that b is still alive.
}
```

Solution to Mistake #5: g_clear_weak_pointer()

```
static void
a_dispose (GObject *object)
{
  A *a = (A *)object;
  g_clear_weak_pointer (&a->b);
  G_OBJECT_CLASS (a_parent_class)->dispose (object);
}
```

Mitigating Memory Safety Errors

- Sandboxing
- Toolchain hardening
- Static analysis (scan-build, Coverity, etc.)
- Dynamic analysis (address sanitizer, other sanitizers, valgrind memcheck) **with code coverage**
- Fuzzing (for parsers)
- Enable assertions in production (except slow assertions)
- Use `g_log_set_always_fatal(G_LOG_LEVEL_CRITICAL)` (except in libraries)
- Consider Rust or other safe languages

Supply Chain Security

Be Careful with Bundled Dependencies

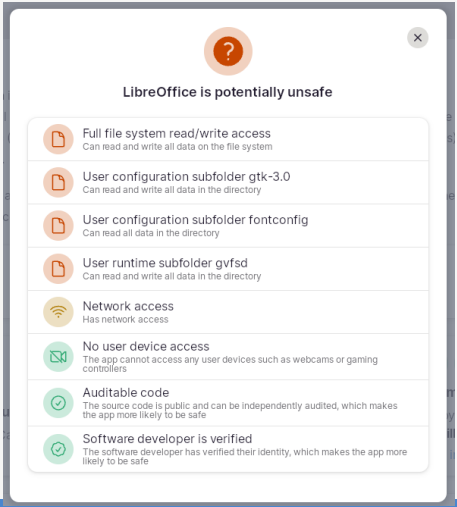
- Your application is only as secure as its least secure dependency
- Memory safety doesn't matter if just one dependency is malicious
- Our Rust apps have too many dependencies
  - glycin-loaders has 286 cargo dependencies
  - librsvg has 283 cargo dependencies
  - loupe has 258 cargo dependencies
  - snapshot has 266 cargo dependencies
- Rust developers: please talk to other Rust developers about this problem!
- A CVE in a shared library requires one update to fix. Same CVE in a static library requires updating everything separately.

Sandboxing

The Flatpak Sandbox is Good

- Sandbox is a contained environment constructed to ensure a compromised application can only do bad stuff within the sandbox
- Sandbox escape is required to harm the host system
- Sandboxing is a defense in depth layer and never a primary security mechanism
- Sandboxing does not excuse security bugs or outdated dependencies

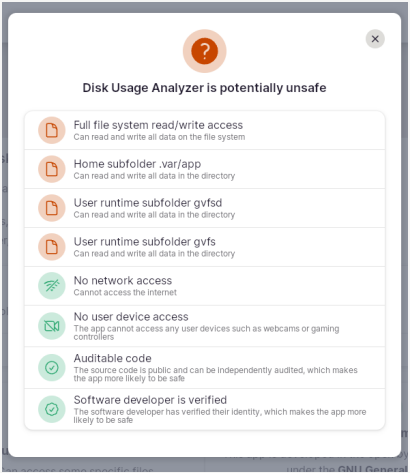
Typical Example: LibreOffice



Why Do We Keep Subverting the Flatpak Sandbox?

- It's been 6 years since flatkill.org first complained the Flatpak sandbox is a lie. It's still true today!
- User trust in trust the Flatpak ecosystem is currently misplaced.
- We must collaborate on portal development instead of punching more sandbox holes.
- Flathub needs to make unsafe permissions (e.g. `--filesystem=` or `--talk-name=`) much scarier.
- Eventually we should delist of applications that use unsafe permissions.
- But we also need a strategy for applications that legitimately cannot be sandboxed.

Counterexample. . .



Call for Action

- Distro maintainers: consider shipping applications using Flatpak
- Flatpak manifest maintainers: keep dependencies updated using flatpak-external-data-checker
- Rust developers: talk to other Rust developers about dependencies
- C/C++ developers: use static and dynamic analysis tools; enable assertions in production