Secure Programming for the Desktop

Jiri (George) Lebl

Why security?

- GNOME gaining acceptance and thus will become a target of security attacks soon.
- It's better to start thinking about this now then when we are widely deployed.

Types of attacks

- Remote vs. local. Many purely local problems become remote with email.

- Escallation of privilages: the “attacker” is able to do something that he normally has no privilages for. For example gaining shell as a user on the machine without having an account.

- Data loss: the “attacker” is able to cause data to be inadvertantly corrupted or deleted.

Types of attacks (cont.)

- Denial of service: the “attacker” manages to force an application to refuse to work, thus effectively rendering the application useless.

- Information leaks: “attacker” is able to gain some information he should not know.

Two Guiding Principles

- *Paranoia* [*]

- *Simplicity* [†]

[*] Should be taken with a grain of salt.

[†] This is really part of paranoia.

Security vs. User Friendliness

- User friendly and security are not exclusive.
- Users do the easy thing, so only easy to use software is secure. Example: encryption, scp vs. ftp, etc...
- Unfriendly paranoia leads to denial of service, or people using an unsafe alternative. Example: ssh

Misc. General Paranoia

- When to trust code external to your application?

- Study semantics of the functionality (documentation / source code)

- Handle errors, don't assume everything just succeeds.

Misc. General Paranoia (cont.)

- Global variables / states.
- Try to keep code compartmentalized and self contained.
- Example: caching values, optimizes speed but allows subtle bugs and the supporting code must be sprinkled around the application.

Input Checking

- Q: When to trust external information? A: Never.
- Sources of external information are: eMail, webpages, documents, the user GUI, configuration, drag and drop data, etc...
- Always have the “public terminal scenario” in mind.
- Check that the information is in the correct form and is actually something that you expect.

Input Checking (cont.)

- Sanity limits for size or complexity of input.
- Sanity limits should be larger then anything really useful.
- Alternatively allow cancellation.
- Both memory and CPU are affected.

Buffer Overruns

Best seen by example:

```
char buf2[] = "DEF";
char buf1[] = "ABC";

puts (buf2);
strcpy (buf1, "123.456");
puts (buf2);
```

will first print “DEF” and then “456”. But worse things can happen (stack smashing, executing different commands, etc...)

Avoiding Buffer Overruns

- Don't use C/C++ (easier said then done)
- Use dynamic arrays and strings (e.g. `GArray`, `GString`)
- Use the helper string functions of glib such as `g_strdup_printf`, `g_strconcat`, etc...
- Avoid pointer arithmetic.
- Allocate and reallocate things on heap rather then "fiddling" with strings on the stack.

Avoiding Buffer Overruns (cont.)

- Have a well defined application wide policy for memory management.

- Don't keep around pointers to data you free. Initialize pointers with `NULL` and set them to `NULL` when you free what they pointed to.

- It's better to reallocate data in memory then to have to manage what the lifetime of some data is, example follows:

Example of Subtle Data Lifetime Bug

```
GtkWidget *dlg, *entry;
const char *str;
...
str = gtk_entry_get_text (GTK_ENTRY (entry));
gtk_widget_destroy (dlg);
...
foo (str);
```

Executing Commands and Shells

Wrong code:

```
const char *s;
char *cmd;
s = gtk_entry_get_text (GTK_ENTRY (entry));
cmd = g_string_printf ("frobator %s", s);

system (cmd);

g_free (cmd);
```

Executing Commands and Shells (cont.)

Fixed code (note the ‘--’ and the fact that the shell is not used):

```
const char *s;
char *cmd, q;
s = gtk_entry_get_text (GTK_ENTRY (entry));
q = g_string_quote (s);
cmd = g_string_printf ("frobator -- %s", q);

g_spawn_command_line_sync (cmd, NULL, NULL,
                           NULL, NULL);

g_free (cmd);
g_free (q);
```

Even better may be to use `g_spawn_sync` and avoid the quoting.

Executing Commands and Shells (cont.)

- Don’t use shell unless really, *really*, needed.
- Quote things properly.
- Think of options (add ‘--’ if needed)
- Synchroneous execution hangs your application.

Temporary Files

- Don't use `/tmp` if you don't need to.
- The "attack" here is to create a symbolic link which you will overwrite.
- Use commands like `g_mkstemp` or `g_file_open_tmp` which do the correct thing.
- Never expect a filename to be available.
- Don't expect things to stay around in `/tmp`.

Opening Files for Writing

Can be similar as /tmp since shared directories might be used.

```
GnomeVFSHandle *handle;
GnomeVFSResult result;

gnome_vfs_unlink (uri);
/* Can ignore errors from unlink */
result = gnome_vfs_create (&handle, uri,
                           GNOME_VFS_OPEN_WRITE,
                           TRUE /* exclusive */,
                           0644);
if (result == GNOME_VFS_OK) {
  g_assert (handle != NULL);
  /* File is opened successfully */
}
```

Denial of Service

- Happens when the desktop “breaks” and can’t be fixed by the average user (non-expert GUI only, no command line, no gconf-editor).
- Printing an error to `stderr` and calling `exit(1)` when something is not kosher is a denial of service.
- Applications, especially the core desktop should work even in very broken situations to allow for repair.
- If an error can be repaired by the program automatically it should just do that. Annoying example: old netscape lock files.

Denial of Service (cont.)

- The configuration or startup state can lead to a crash which causes a denial of service.
- It's needed to detect a crash on startup and allow the user to reset settings, load files one by one or otherwise get into a working state.
- Example: broken Nautilus thumbnail/preview plugin crashing nautilus viewing the home directory.
- Moral: expect bugs, expect crashes, try to mitigate the damage they may do.

Denial of Service (cont.)

- Logfiles can fill diskspace which may not be reclaimed until user logs out (files are only truly deleted when they are closed).

- Output to `stdout`, `stderr` is logged!

- Avoid spurious output of errors especially in response to external data to `stdout` or `stderr`.

- Set a maximum of errors per document or per interval of time when printing to `stdout` or `stderr` or to a log file.

Information, Cookies, Authentication and Random Numbers

- Use encryption when possible.

- Modern encryption methods should allow this to be totally transparent to the user.

- Authentication may require some setup on the part of the user, try to keep this as simple to do as possible so that people don't go to unsafe alternatives.

- **NEVER** home cook new protocols for encryption. Use well tested and scrutinized protocols and libraries such as OpenSSL.

Random Numbers for Security

This section doesn't apply for random numbers that don't need to be secure

- Pseudorandom number generators don't generate random numbers.
- Use `/dev/urandom` directly if possible.
- Pseudorandom number generator will not take a non-random seed and make it random.
- Once you have something that has some entropy, no need to further massage it unless you need to use less space (then use MD5, SHA1, not `GRand`)

Random Numbers from Current Time

Sort of going off topic but I'm a maths student so I wouldn't feel right if you weren't subjected to this.

```
GTimeVal now;
g_get_current_time (&now);
```

Using `now.tv_sec` and `now.tv_usec` gets about 32 bits of entropy with 68 minutes uncertainty. For a 32 bit number use

```
(now.tv_sec << 20) ^ now.tv_usec
```

and not

```
now.tv_sec ^ now.tv_usec
```

How Many Bits in a Cookie?

I can count from 1 to 2^{32} in a busy loop in 5 seconds. If computer speed keeps doubling every 18 months, I'll be able to do the same with 2^{128} in about 150 years. Conclusion: 128 bits is enough and will likely be always enough. Use 196 or 256 if you are truly paranoid.

Summary

Remember: *Paranoia*, *Simplicity*

Full paper is online at: `http://www.jirka.org/`