Secure Programming for the Desktop

Jiri (George) Lebl

Abstract

While desktop applications are usually not thought to be on the "front lines" in terms of security, they can in fact pose just as much of a security threat as a server application. In fact this threat can be even greater as desktop computers are usually not as well managed as servers and thus malicious activity can go undetected much more easily. Furthermore as recent Windows worms have shown, the fact that there are a lot more desktop computers than servers makes them a much more interesting target for many malicious activities. When we put the filesystems on the network and perhaps even have multiuser machines, then we even start facing the same problems that servers are facing. While this threat is very real, desktop applications tend to be relatively lax about various potential problems. In this paper I will describe several different issues that tend to arise and that I've encountered and how to avoid them. These issues range from simple denial of service (DoS) attacks (which waste productivity and IT budget when we start moving onto corporate desktop), through information leaks, to actual break ins. Focus will be given to GNOME and C related issues, but in general these ideas can be applied to any desktop and programming language.

1 Introduction

As GNOME gains wider acceptance on the desktop, the issue of security will start to be more crucial. One of the reasons why people are moving to a Linux/GNOME desktop is the security as compared to Windows. We have now a greater chance of "getting it right" before GNOME becomes widely accepted instead of then fixing things as large scale "attacks" happen such as is the norm in the Windows world currently.

So what are the potential issues that arise in desktop software? Firstly there is the distinction of remote versus local issue. If a problem can be exploited remotely then it is a far greater threat. Fortunately this is not usually a case for desktop software as desktop software usually does not present any remote interface. Unfortunately because of email and easy file sharing, a local problem can quickly be turned into a remote problem with perhaps just a tiny bit of cooperation of the local user.

The "attacks" on desktop software could then be classified as escallation of privilages, data loss, denial of service and information leaks. Escallation of privilages is when a user is able to either do something they do not have privilages for or when a user is able to execute a command as another user. We can also consider a remote break in (perhaps gaining shell as a user without having an account) as an escallation of privilage attack. Data loss is when the "attacker" manages to destroy some valuable data, perhaps making the user inadvertently overwrite some data. Denial of service is an attack where the "attacker" somehow manages to force an application to refuse to work, thus effectively rendering the application useless for the user. Information leaks are when an application divulges some private information in some way. Information leaks can then be used to mount other attacks, for example if the attacker is able to find a password or a cookie that would be needed to log in as a user.

2 Two Guiding Principles

This paper is more of a series of issues that I encountered or seen rather than a well thought out and self-contained step by step guide to making secure applications. One of the reasons for this is that security of any application is case by case. I will give what I consider are the two main principles behind safe and secure code. Obviously these should be taken with a grain of salt as they are subjective, and requirements vary from application to application.

The main guiding principle in writing secure software is to expect the unexpected. That is the number one thing you should take away from this paper is:

Principle #1: *Paranoia is good.*

And here I don't mean paranoia in the sense of testing if the martians landed after every function call. What I mean here is that you should always think about any error conditions that may arise even

if they seem unlikely, and further that you should not blindly assume certain things are true. Paranoid code is not necessarily more complex code. Simplifying code is often also the "paranoid" approach. That is, if you can do something in a simpler, easier to understand, but perhaps not as optimized fashion, then it is less likely that there will be a bug. This leads to the second principle.

Principle #2: *Simplicity is good.*

By this I don't mean that you should not check error conditions. Doing that obviously makes your code shorter, but not more secure. What you should take this to read is that unnecessary optimizations or unneeded features or unneeded complexity leads to more security problems. This is really sort of a subprinciple of principle #1 in my view as striving for simplicity is really a "paranoid" way to code.

Security issues are usually not found by testing the software, but by looking at the code. The less code that there is, the less code to verify, and the simpler the code is to read the simpler it is to verify it (note that simpler code to understand need not necessarily be less code).

If your coding style is more paranoid than necessary, then most likely nothing bad will happen. On the other hand if you are not paranoid at all, then it is likely nothing bad in terms of security will happen either, most issues will not be exploitable in any meaningful way on any sort of standard installation. But if you take this view, then security issues will eventually arise. Somebody at some point will figure out how to exploit some overlooked issue in a way that hasn't been thought of before, where paranoid code might have avoided this. If we knew all the possible security issues that might arise, then we would not need to be as paranoid in coding to produce secure applications.

Simplicity should really translate into all parts of the application. It is better to have a smaller amount of well tested, well behaved code that checks for all possible errors than to have thousands of options and features translating to thousands of different code paths which will likely never be tested or verified in all the possible ways that the application is used. Obviously an application that doesn't do absolutely anything is the most secure one. But also not a very useful one. You should however concentrate on what the application is supposed to achieve for the user. Since this is already somewhat of the guiding principle of usability, you should not find too many conflicts between usability and security.

3 Security vs. User Friendliness

There is a common misconception that security equals applications that are not user friendly and vice versa. To some extent this can be true, but in vast majority of the cases it is not. Just the opposite. Dealing with security problems in a non-user friendly manner can introduce new security problems such as denial of service. If the user cannot get out of a problem without help, it's just as if the program was not working at all.

Biggest problem here is making things too complicated to do correctly. If the "correct" (read: secure) way of doing something is hard to do, users will not do it the "correct" way. One example of this is email encryption. Currently solutions such as pgp are so incredibly hard to use that only people who wear pocket protectors, thick glasses and run debian actually encrypt or sign their emails (I also assume that NSA people use encryption). Unless you know how all this stuff works it is currently very hard to use and so no one does it. Another example of this idea is ftp. It is far easier to use ftp than scp/sftp for reasons of all the scp/sftp GUI methods sucking very badly (all the ones I've seen, Nautilus is semi-there, but not quite). So people will tend to use ftp to transfer files. People underestimate security risks, so they will pick the solution that's easiest to use and not necessarily the most secure. Education doesn't work, what works is making it easier to do things securely.

Some people writing security software are incredibly paranoid. To these people, maximum security is more important than the fact that most of the world will not use their software and will instead use some insecure garbage. For the desktop to be secure we must have users that actually use secure software. If an insecure program happens to be more convenient, it is likely people will use it. We don't need to care about NSA being able to break into people's computers. We want to care about Bob from down the street sending your mom a virus that makes her computer into a spam relay and puts lots of kiddie porn on her hard drive.

An example of unfriendly paranoia is ssh. If something goes wrong, such as a host key changes it tells you to do stuff such as edit a text file. This obviously makes ssh unusable from the GUI even though it can be used for remote execution of GUI apps, it is only useful for that if set up correctly and other less secure methods are easier to use. Another example from the ssh camp was an ssh application for the mac, "Nifty Telnet" I believe or some such name. It had a

GUI, but by default it wanted to use telnet. It obviously had ssh capability, but for some reason I could not make it connect by ssh in several minutes of GUI twiddling. So the easy mode of operation was unencrypted telnet and that's what most people would then use. And since the tag line for the application was something like "secure communications for the mac" (again paraphrasing, I don't remember it exactly), the user would be fooled into thinking that it's actually secure, making matters even worse.

4 General Paranoia

Firstly let's look at using external code to your application. It is usually better if you can use some library which already implements some functionality than reimplementing it yourself since the library has most likely already been tested and verified by its authors and other users of the library. But on the other hand this only holds for well used libraries. If the library has only a few users it is a lot less likely that it is well tested and you may be better off writing your own code if you don't need too much of the functionality. There's always a tradeoff here. It also depends on your application and on what the library does.

If you do use an external library it is always good to look at either the documentation or the source code carefully and note all the semantics, especially all the things that could go wrong. Many times assuming that some call succeeded without checking leads to bad security problems. For example recent KDM/XDM had a problem of not checking the error result of a pam (library for authentication) function and that led to a possibility of gaining root privilages. A thing to keep in mind as well is that a function could also be broken (for the user may be using a different version of the library than you are where there is a bug). It is not terribly important but if you keep this in mind and code defensively, either by putting `assert` calls in your code or taking other precautions you can minimize the effects of a bug in the library.

Another general paranoia issue is avoiding global variables or global states or such. It is best if your functions are as self contained as possible as that will make them far easier to verify. An example of this is caching. Sometimes it may be useful to cache a result of a certain calculation if you ask for this calculation many times. However this is an optimization that may lead to very subtle bugs. The problem is when something happens that would affect the result of the calculation, you must forget about the cache. This means that the code will not be self contained, but parts of this functionality will be sprinkled about all around the application wherever things may change. This for example bit me in GDM where I had an integer that counted the number of active sessions. A bug in the code allowed for this number to not reflect reality resulting in a possible denial of service attack. The fix was to replace the global integer with a function which just recounts the number of active sessions. Now we do this several times right after each other in places, but it's never an actual problem of performance and it's always correct. Further there's no need to both update the list of active sessions *and* the number of sessions. The moral of this story is thus to avoid caching anything unless you really positively know that not doing so will very adversely impact performance.

5 Input Checking

A prevailing theme behind secure software software is to never trust any information coming from the outside of the software. Or at least don't trust it as far as not trusting it is reasonable. That is, information coming either from the user, from configuration, from other programs, from files, etc... should always be checked before used.

The reason for checking what comes from files is made obvious by the mail virii so prevalent recently. This means that you should not assume that a file in a format that you understand was created by your software. It could have been hand crafted precisely to break your software.

The reasons for carefully checking what the user gives you through the GUI are several. You may argue that the user will never input anything malicious into the software since at most they'd be "cracking" their own account. But if the malicious input is complex enough such as a long URL, the user may be fooled into cutting and pasting that into an input. And of course if the computer is perhaps a public terminal, and the user sitting at the keyboard may be the malicious user.

Now sometimes it would be "legal" to have some part of the input be either arbitrarily large or arbitrarily complex. But perhaps it would not be reasonable for actual use of the software. It is best to set up some reasonable limits. These should be larger than anything that a user would use in an actual usage, but such that for example we don't unnecessarily exhaust memory. As an example I will give a bug that crept into the GDM socket protocol code. I was very careful about checking for buffer overruns (see section 6) but I forgot about one thing. I read input line at a time into a resizable string (which would resize itself

according to the size of the input). But I forgot to give a limit on the size of each line which led to a possible denial of service attack by forcing GDM to fill up the memory and perhaps crash. Now given the protocol it is unreasonable to expect more than say 256 characters of input, so we can just read at most that much and discard the rest of the line or give an error or something similar.

Similar issue can come up with reading any file. For example if you keep putting stuff from a file into a `GtkLabel` because you don't expect the file to be long or some field in the file to be long, I can send you a long file with a lot of text in that field and you will be very surprised at how quickly `GtkLabel` eats memory when you display an entire play from Shakespeare in it. Now the user may not realize what's happening and by the time they do they have a heavily swapping system that may be hard to regain control of. Perhaps the "leak" causes some other piece of software to crash or possibly exploits a security hole someplace else. Or if I, as an evil person, can somehow manage to convince some other (non-evil) person to open such a file in some way such that it is opened every time the program is run, I've got a successful denial of service attack (see section 10). Another way to deal with this, instead of a hard limit, is to allow easy cancellation of a long operation.

Similarly as filling memory, I could also hog the CPU. It is conceivable that I could cause some sort of long or infinite loop in the application by specifying some unreasonably large parameter as input. For a long time for example Windows had this nasty habit of taking up all your CPU if you connected to the right port and typed some garbage. Similarly one could annoy the admin of a terminal lab by typing some unreasonable number into some application on all the terminals and rendering them slow or useless.

6 Buffer Overruns

The most famous security related bug is the buffer overrun. The basic idea of this is that the program at some point is forced to unwittingly write in some area which it did not reserve. Usually this happens when you perhaps allocate an array of some many bytes but the code then writes some input into this array without properly checking if it will fit. So what will happen is that this input will overwrite other things in memory which may have different uses. An example is the following code

```
char buf2[] = "DEF";
char buf1[] = "ABC";
puts (buf2);
strcpy (buf1, "123.456");
puts (buf2);
```

When you run this, it will first print out "DEF" and later "456", even though both times we print `buf2` and we only modify `buf1`. So if `buf2` was a command to execute that you thought was safe to execute we could perhaps write `rm -fR /` instead of `456` into it. Other things can be modified as well other than strings. For example we could modify where the function returns when it's done if the string or array is on the stack as it is above. The most vulnerable is therefore stack itself, but things allocated on the heap can also be vulnerable.

This is of course an issue only in languages that allow you this low level access to memory such as C or C++. So one way to avoid this problem completely is to just use a language such as Java, C#, Python or other such languages. But of course that's not always an option so let's see what you can do in C to avoid such problems.

Firstly you should try to use dynamic arrays and strings when possible (`GString` and `GArray` in glib), or the helper string functions of glib. One such useful call is `g_strdup_printf`, as a replacement of `sprintf`. It will always allocate a large enough string before "printing" into it and so you will never have to worry about having to allocate enough space and getting it potentially wrong. Also it is a good idea to just avoid any pointer arithmetic or fiddling with strings "by hand." Best to always allocate a new strings and use one of the string helper functions from glib. While allocating and reallocating strings all the time may be more time and memory consuming, it is generally preferable to have simpler, correct code rather than fast code. Also allocating things on the heap rather than using the stack makes certain attacks not possible (the stack smashing attacks where we overwrite the return pointer for example). You pay the price of a little slower code but in most cases this won't make one bit of difference to the user.

It is also a good idea to standardize on a single way of handing memory allocation in your application and stick to it. I will give you my own guidelines for strings that I use that are in my opinion the best way to handle the situation.

Firstly when passing a string to a function I always consider this string as a constant inside and never modify it. If I need to modify it I will make a local copy of it inside. If I need to return a modified string I returned a newly allocated copy. For example here is some pseudocode to give you an idea:

```
char *
a_function (const char *str)
{
  char *str_copy = g_strdup (str);
  ...
  return g_strdup (...);
}
```

The function never messes with anything in the data owned by the calling code and vice versa. If the function does some fiddling with memory and strings, then both the allocation code and the code that does the "fiddling" is right there together and can be checked much more easily than if the memory allocation is at one place and the "fiddling" in another. And since we return a newly allocated string the calling code will now be able to decide on the lifetime of this, so there is no need to make sure that the calling code doesn't touch it for some time. There are many examples in GTK+ itself which don't follow this practice so it's important to know these. Here's an example of a very subtle bug where we will have an invalid pointer (this exact case is not a buffer overrun and something much harder to exploit, but pretty subtle and still an issue). Imagine you have a dialog, say `dlg`, and a `GtkEntry` inside the dialog say `entry`, and you wish to look at the text:

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

The function `foo` gets a pointer to freed memory, which could by now be allocated for some other purpose and thus contain something different. The fix would be to use `g_strdup` on the string before the dialog is destroyed as we don't want to have pointers into the dialog object that is just about to die.

Thus it is good to have clearly defined "rules of engagement" for your application, whatever they are. It should be clear which code "owns" a particular piece of data (memory) and what is the lifetime of this memory. A good practice I use is to zero pointers on initialization and after freeing them especially if there is perhaps some code after the `g_free` code that could perhaps use the pointer. Here's an example:

```
char *str = NULL;
...
str = g_strdup (...);
...
g_free (str);
str = NULL;
...
```

Trying to dereference a null pointer is a crash that is not exploitable, trying to dereference a pointer to freed data (or a pointer to some random place) is not always a crash and can lead to security issues under some circumstances. Similarly if I'm freeing a `GList` I try to set the `data` field to `NULL` for the same reason.

7 Shells, Options and Executing Other Programs

One of the more subtle ways of how to break security is to use the shell to execute something. Best rule of thumb is to never ever invoke anything that executes the shell, especially if any part of the executed string comes from external sources (configuration, user, document file, dnd data). For example the following code is very unsafe:

```
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);
```

Now the if the string turns out to be something like "`blah > ~/important.file`", then the `frobator` command will only get "blah" as an argument, and `~/important.file` will be overwritten.

The simplest (and least paranoid, and furthermore still wrong) way to solve this is to use `g_shell_quote` which quotes the string for the shell such that when the shell sees this it will not interpret the string at all. Internally it will quote the string with single quotes and make sure to properly escape any single quotes that may appear in the string. So the code may look like:

```
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);

system (cmd);

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

Now suppose that the `frobator` command can take an option such as `-o=some.file` to redirect output to `some.file`. So now the string happens to be guarded from the shell, but the `frobator` application can still interpret it wrongly and do damage. The solutions are several. Firstly, if it never makes sense for the

string to be passed to frobator to begin with a dash (such as if it is a uri which then begins with the scheme string which is never a dash), then perhaps you should check for the initial character being a dash and in that case report an error to the user. Alternatively if such strings make sense and the `frobator` command has the `--` option which makes it treat all following arguments and non-options, then you should use that and replace the command as follows

```
...
cmd = g_string_printf ("frobator -- %s", q);
...
```

But here you need to make sure that all implementations of the `frobator` command support support this option. If not you have to use some other method or perhaps disallow such strings. If the string is to be a filename a safe way to do this is to prepend a `./`, though here you should really use not a slash but the `G_DIR_SEPARATOR` constant for sake of portability.

But note that in the above we aren't really using the shell for anything. So the safest and most paranoid way would be to use `g_spawn_command_line_sync` or directly use `g_spawn_sync` (see the documentation on those two). For example replace the `system` call with

```
...
g_spawn_command_line_sync (cmd, NULL, NULL,
                           NULL, NULL);
...
```

Obviously in the above we should also check for errors and behave accordingly. It is always the most paranoid and safest route to avoid shell if you possibly can. At least avoid it where it is not used anyway such as above.

Further thought should be given to actually executing commands synchronously or not. This is because if you do this synchronously, then your application will hang until the command completes which is not very good if the command could hang or if it could take a long time. It may perhaps be good to execute it asynchronously. But with that you are opening a whole bag of further possible bugs that could turn out to be security issues. You suddenly need to manage this process (figure out when it completed and such), and you need to make sure usually not to execute it again while an old incarnation is still running (this depends on the nature of the command). Further you are getting into the land of possible races, so it is a tradeoff.

8 Temporary Files

Problems with temporary files got a lot of coverage a few years ago. Basically the problem is when you are writing to an area where other people have permissions for writing. Those other people could be malicious and if they know what filename you are going to use, they may redirect your application with a symbolic link to a file that they want to destroy that only you have write-permission to. So the first rule is to have non-predictable names in `/tmp`, and second is to only use atomic, non-destructive functions. Only `root` would be able to move other files there, so you cannot expect a certain filename or directory name to be available to you.

The best way to avoid problems with `/tmp` is to avoid using `/tmp` if you can. Sometimes you can use the home directory instead, perhaps `~/tmp`, but be aware that home directory can be non-writable on certain occasions, so you need to provide a workaround for that.

If you must use temporary files or directories in the `/tmp` directory, use `g_mkstemp` or `g_file_open_tmp` (or `mkstemp` if you don't use glib) for creating files. This function will create a unique name for the file, open it and give you a file handle that you can use for writing. This must be done in one step, if you'd only check that the file doesn't exist and later open it for writing, someone might beat you to it and create the malicious link.

To create directories, you could use code like

```
GRand *r = g_rand_new ();
char buf[256];

errno = 0;
do {
  g_snprintf (buf, sizeof (buf), "/tmp/foo-%x",
              (guint) g_rand_int (r));
} while (mkdir (buf, 0700) != 0 &&
         errno == EEXIST);
g_rand_free (r);

if (errno == 0) {
  /* only here can we use the new directory */
}
```

There are several things to note about this snippet. Firstly `mkdir` will check the name and create the directory all in one go, atomically, and is thus safe (it will not overwrite something if the name exists). The reason why we use a separate `GRand` object rather than just using `g_random_int` is general paranoia. Someone could have gotten hold of the global random buffer in some other way, or maybe they can use this to figure out the global random buffer. Third

thing to note is that we don't just check for failure of `mkdir`, but also for the type of failure. If for example the system is badly set up and we don't have write permission to the `/tmp` directory we'd go into an infinite loop instead of catching this if we didn't check for the proper `errno` value. Finally, since it is possible to drop out of the loop without having the directory ready, we should handle the possibility of an error. Later in the lifetime of the program, you should always assume that the directory may have disappeared, perhaps by the sysadmin or some automated cleanup script, so don't store anything critical there for any long period of time (note that a program could be running on some users desktop for months at a time, so this is a real issue). To avoid automated scripts removing the directory, you can touch the directory using the `utime` library call say once a day (just make a timeout in your app that does that). Then automated scripts will notice the timestamp and leave the directory alone.

You should always create a directory for yourself if you will use temporary files a lot, it saves on code later and there will be less chances to screw up. Once you have a directory that only your user can read, write and execute, then you no longer have to be careful about filenames inside this directory.

Final thing to note about temporary files is one of their positive qualities. The `/tmp` directory is pretty much always on the local filesystem (sometimes as a ramdisk). This means that data written to it are never transferred over the net with say NFS etc... So for example cookies for various things might find a much better home in `/tmp` than in the home directory for this very reason, though the caveat about `/tmp` not being a place to store permanent data still applies here. Furthermore the `/tmp` is always pretty fast and the home directory may be very slow if remote.

Final note of caution is to avoid the use of the `TMPDIR` environment variable. Historically this pointed to the place that you should use instead of `/tmp`. However this becomes a problem for setuid or setgid applications where the environment is controlled by a different user than what the application runs under, and this could be abused. Also since so many applications don't use it, it's not practical to use anyway. It's just better policy to hardcode `/tmp` which leads to simpler code and avoids the potential for problems.

9 Opening Files for Writing

Now let's look at a related issue of opening files for writing. A similar thing that is happening with the `/tmp` directory may happen. If the user is saving a file in a location that is not owned by him (say for example `/tmp` or some shared directory) then another evil user could yet again set up a symbolic link and mess with our good user if he (the evil user) can guess what the filename may be. This may be easier than you think; humans are bad at making up random filenames, and plus may be fooled into saving to an existing filename that the malicious user has created (he created a malicious link).

Thus to protect against such things it is best to avoid doing stuff such as just opening a file for writing, but instead may be advantageous to first unlink the file (which is still OK to do if the file doesn't exist) and then create a new file in exclusive mode. With `open` call, use the `O_CREAT|O_EXCL` flags. If you are using gnome-vfs as you should for a GNOME application, you can use

```
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 */
}
```

The exclusive flag will basically make the call fail if the file already exists. So if someone is playing tricks with us and causing a race condition to occur, it will be caught by the exclusive flag. Obviously then you need to handle this error and alert the user.

10 Denial of Service

This is the least destructive security issue, but it is also the easiest to suffer from. When we move onto the desktop of people that are not UNIX experts and cannot "fix" the issue without use of the GUI which is being DoSed, then it can be potentially destructive for such users, either having them lose time and money to have it fixed by someone, or even losing data if they just reinstall the system carelessly perhaps.

The most evil denial of service problem is usually being done in the good name of security paranoia, and that is to just die, print something to `stderr` and let the user figure it out. For example suppose that some permissions are bad, then broken software

would just die (there are such issues in gnome-session for example, see bugzilla). Sometimes these issues come from external libraries such as libX11 and libICE, which obviously come from the time when all the users were UNIX gurus and launched GUI programs from the command line.

If the program can work in even some limited way without some prerequisite, it should at least run in this limited way. This is especially true for things one would need in the GUI to repair the issue, for example gnome-session, Nautilus, the panel or GDM (or the equivalents in any desktop). The user should always be able to log in and do some basic things such as file management and editing and whatnot even in an extremely broken state. This is because we're trying to make sure that people don't log in as `root`, then a system or user account should be repairable by GUI from a user login.

In the case that the error can be repaired automatically, perhaps there's not even a reason to bother the user with it and just repair it. For example Netscape used to require you to remove a lock file whenever it crashed. Since that is something that the software can repair itself, it should not force the user to suffer for its own bugs.

Second type of DoS problem is a bug in the software that causes a crash on some set of settings, or on loading a certain file. This is especially true if the offending file or data or whatnot is then loaded automatically for some reason. The attacker can then just somehow make the user either load that data or use that setting. An obvious example is when a mail reader crashes when displaying the subject line of an email, then the user can't ever check his email again. But other problems can be abused as well. For example if I don't like the IT guys in some company I could mass mail the employees with "if you want your desktop to be even cooler, check the frobinator setting in the control panel," given I know the frobinator setting makes the session crash on startup. The simplest way to solve this is to write a temporary file to disk (in the home directory say) and remove it on exit. Then next time you run and find this file (you have to check that it's not another instance that owns it) you tell that to the user and offer to start with a clean slate, perhaps offering to reset the configuration to some default set, which being the most tested configuration, most likely works.

Such a denial of service attack was (and perhaps still is) with plugins for Nautilus. I was actually hit once with a bug like this. Some sort of plugin was trying to make a preview or thumbnail or whatnot of a flash file I put in my home directory. Unfortunately it crashed whenever it tried. Which meant that Nautilus crashed whenever I moved into my home directory. It took quite some time to figure out what was happening. Obviously this is the fault of both the plugin and Nautilus. Nautilus needs to be more careful and robust with respect to buggy plugins (they will always exist) and the plugin needs some way of perhaps remembering that it crashed on this file (with the method described above). That way it can tell me next time I try to look into the directory and I could fix my system and file the appropriate bug. The way it was, all I saw was a fairly random crash in Nautilus with a useless stack trace and had no way of knowing what went wrong. Furthermore since the standard way of starting Nautilus opens the home directory I couldn't really launch Nautilus at all.

So the moral of the story is to expect crashes. It is to expect that your code can crash with certain settings and allowing the user to reset settings or another way to recover without making him or her find the problem and fix it. For example it took me a *long* time to figure out what was wrong with Nautilus in the story mentioned above, even a seasoned admin person would have trouble figuring it out quickly.

Finally there are denials of service which take over CPU, fill up the memory or the hard disk. For CPU and memory, the idea is to put sanity limits on both the amount of work or memory you use especially in response to data coming from the outside (including even such things as configuration). We have talked about this in the input checking section (see section 5). So let's look at filling up the disk. Sometimes this may not look like you are doing something dangerous. For example just printing something to the standard output. But note that most of the time this is redirected to some file. Newer GDM in default configuration will intercept this output and has a sanity cap on this of a few hundered kilobytes, but unfortunately some distributions may ignore the default GDM session setup and just re-redirect it to a file again reintroducing this problem. The idea is to force a program to repeatedly print some error. For example imagine the web browser finding an error in a web page and printing out such on the standard output (or standard error), now suppose I'd be my malicious self and create a web page with lots and *lots* of these errors, then anyone coming to my web page would have his home directory filled up with random garbage and would have to relogin to continue working (note that removing such a file does not work since it's still open and will thus exist on disk even after being deleted until the session is over). Now suppose that either the login manager is not GDM or

some other login manager that knows of these issues or the session decided to log it to some file that the login manager does not know about. It is then possible that the user will not be able to log in. The problem is not restricted to web browsers. It can be found in any document driven application that is told to open some document (which may come from a malicious source possibly through email). The best way to deal with this is to put a sanity limit on the number of errors or warning you report for any document. Nobody is likely to read them anyway if there is more than a couple of thousand of those. Another way to deal with this is to throttle these messages rather than just capping them, say only print 20 inside any minute or some such other limit.

Similar holds for any log files you create, be careful not to overrun those. Most likely these are going into the users home directory for a desktop application and a full home partition (or filled up quota) could lead to anything from not being to log in to data loss (automated script deleting files from over quota account).

11 Information, Cookies, Authentication and Random Numbers

What would a security paper be without at least mentioning random numbers, encryption, authentication and related issues. Most desktop applications don't have to deal with this, but a few do, such as networked games, web browsers, mail readers, etc... Let's first talk about encryption. Any application that talks over the network should attempt to use an encrypted channel. It should also never be the default to use an unencrypted channel if that is possible. With modern cryptographic techniques, there is no need for the user to have to fiddle with anything to have an encrypted channel. No need to set up keys, no need to set up anything, a secure channel can be set up by a public key cryptosystem or by one of the variants of Diffie-Hellman or similar. Using the elliptic curve techniques this can be done very efficiently so there's no excuse of this affecting performance. If you find that there needs to be an extra step done by the user to simply get encryption, then something is wrong with the design.

For authentication it is a different story. The user must be somehow authenticated by the remote side. Or the user might want to verify that the remote side is really what it claims to be. So both sides must take some step to authenticate themselves, either by setting up an account somewhere or having some certificate signed or something of the sort.

But the most important rule about encryption/authentication is to never, ever, never ever ever, home cook new protocols for this. Use one of the already developed layers such as OpenSSL, and build on top of that. There are many subtleties that arise here, and having to change a protocol once deployed is very tough. Things such as OpenSSL already went through a very rigorous analysis in real life use, so it should be the safest option to use some already developed technology like that.

If you really need to get random numbers for any sort of authentication, you can again use a library like OpenSSL which has facilities for getting good random numbers. Alternatively read directly from `/dev/urandom`. Never use a pseudorandom number generator such as `g_random_int` and friends, and god forbid `rand`. `g_random_int` does have 128 bits of theoretical entropy (randomness) but it gets that by just reading the seed from `/dev/urandom`. You will however never get more than 128 bits of randomness no matter how many times you call `g_random_int`. You can fill 10 kilobytes using `g_random_int` and I will be able to reproduce it exactly just after I guess the 128 bit seed value. So if you'd hope that using `g_random_int` (or `rand`) several times would give you longer and thus harder to guess password or cookie, you'd be wrong.

The algorithm used by `g_random_int` and friends is very fast, seems to produce nice random numbers, but is not well tested and scrutinized. For example it could very well give output of a lower entropy than 128 bits even though it used 128 random bits as a seed. In any case you'd just be inserting a middleman into the equation. `/dev/urandom` should give you quite a bit of good entropy. Note that `/dev/urandom` is also pseudorandom but seeded with actually random data as often as random events happen on the computer, so as long as you don't want way too much random data at one go, you get very good results.

Further note that taking something that is not too random, such as the current time in seconds, using it as a seed in a pseudorandom number generator doesn't get you anything more random. It may just give you a more "random looking" answer, but not something more random, it would just give you a false sense of security. If you are on a system that doesn't have `/dev/urandom` or any equivalent and you can't use any other method and you need a random cookie, then the following code may get you some level of randomness

```
GTimeVal now;
guint32 cookie[2];
```

```
g_get_current_time (&now);
cookie[0] = now.tv_sec;
cookie[1] = now.tv_usec;
```

Now the cookie may be 64 bits long, but you only get about 32 bits of actual randomness and that only if you can't guess the time to within 68 minutes. If you can guess the right second you only get about 20 bits of randomness (the microseconds). You will never get 64 bits randomness like this, you can only get 51 bits of randomness and that only if you can't guess the time at all (that's over 60+ years span since the time is an integer). No need to use `GRand` or any other random number generator to make it look "more random" unless it makes you feel warm and fuzzy. If you actually need a 32bit number and want as much entropy as possible then use the following expression

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

That gets you just under 32 bits of entropy if you can't guess the time to within those 68 minutes. Never use something like `now.tv_sec ^ now.tv_usec`, it looks correct, but you only get 20 bits of entropy for the first 11 days. The thing is that the microsecond field has about 20 bits that change very rapidly, so we'll consider all those random. But the seconds field has the more random bits as the least significant bits. So we want to shift those bits to occupy the place where the microsecond field has all zeros.

If the "attacker" wouldn't have access to the local machine and thus can't figure out the process id and the parent process id, you also get a slight bit of randomness from the `getpid` and `getppid` calls. The amount of entropy you get from these depends on how and when your application gets started and on the number of other processes on the system. Now if you need a smaller number of "bits" and you have a whole bunch of numbers which have some entropy but are not exactly random (such as the time values or the pid and the ppid) you could concievably use these to seed a `GRand` object using the array seeding function and that would act as a sort of "compression" function. The caveat here again is that the algorithm that `GRand` uses has not been well tested, MD5 or SHA1 might be better for such a purpose. In fact if you are going to use the current time up to microseconds and the pid and ppid you might as well just use the standard initialization of a `GRand` object as that is what it will use. If you are going to write something to a file try to put the file in a directory that is not readable by the attacker. That way they cannot check the timestamp on the file to figure out the seed.

However do note that 32 bits just may not be enough nowdays to prevent a brute force attack (depending on the application of course). The XDM (and old KDM) X cookie has 32 bits of entropy (they used `gettimeofday` badly, needing 18 hours to achieve 32 bits of entropy as seed and then `rand` to fill 128 bits, really, I'm not kidding). After testing I found one could break the key to the X session in a day or two and that was assuming they would truly get 32 bits of entropy. If you could guess the time the session started to within an hour you could break it within about 2-3 minutes.

The time for brute forcing certain keys is only going to get shorter as things get faster, but note that complexity here grows exponentially. Using 128 bits will most likely always be safe (for most reasonable definitions of "always"). Use 192 or 256 if you are really paranoid. Anything more and you're just wasting space. On my box an optimized loop that does nothing but counts from 1 to 2^{32} takes 5 seconds to run, so 2.5 seconds is our lower bound on brute force attacks. So for 128 bits it would take at this speed 6280771381458042997966 years. Now if speeds of computers keep doubling every 18 months, we will be able to run this in the time of 1 year in about 108 years or so and in 2.5 seconds in 144 years. And that's only the lower bound. So we have at least around 150 years to go before 128 bits have any chance to become what 32 bits are now. And note that network bandwidth has not grown at such a rate anyway. Thus in conclusion, unless new medical breakthroughs happen, you are unlikely to be around when 128 bits could become a problem.

12 Summary

In summary, security problems can arise in all parts of the desktop. It is important to always have the main problems in mind when writing any software. It is also important about the major use cases for your application and what could be the threats in that situation. It is also important for a desktop application that problems are "fixable" either automatically or through the GUI, without having to resort to the command line or hand editing the configuration database.