

Maemo Diablo Reference Manual for maemo 4.1

Quality Considerations

December 22, 2008

Contents

1	Quality Considerations	2
1.1	Quality Awareness	3
1.2	Secure Software Design	12
1.2.1	Overview	12
1.2.2	Elements of Secure Software Development	12
1.2.3	Threat Analysis	13
1.2.4	Security Testing	17
1.2.5	Conflicts of Interest?	18
1.2.6	Tools and Resources	19
1.2.7	Maemo Security Policies	20
1.3	Maemo Coding Style and Programming Guidelines	21
1.3.1	Maemo Coding Practices	22
1.3.2	Basics of Good Programming	23
1.3.3	Maemo Conventions	25
1.3.4	Power Saving	32

Chapter 1

Quality Considerations

The maemo developer needs to take care of many details, when developing a good quality application for an embedded device. Compared to the desktop Linux, a developer needs to be more careful with performance, resource usage, power consumption, security and usability issues. This document provides a generic test plan for developers to help them to create a good quality software.

Performance and Responsiveness

The user wants to be in control: the device must be responsive at all times, and give appropriate feedback. The UI responsiveness can be achieved e.g. in the three following ways: First, using process events by showing a banner and blocking the UI while operating. Second option would be to slice the operation into parts while updating e.g. a progress bar. Thirdly, threading can be achieved with the help of GLib, but this is increasingly hard to implement and debug. For time-consuming tasks, the second option is recommended. If the operation takes only a second or two, the first option can also be used.

As other general guidelines, CPU over usage must naturally be avoided. This goes also for completely redundant operations, such as unnecessary screen updates. Also, when an application is not visible, it should be inactive to prevent it from slowing down the application user is currently interacting with.

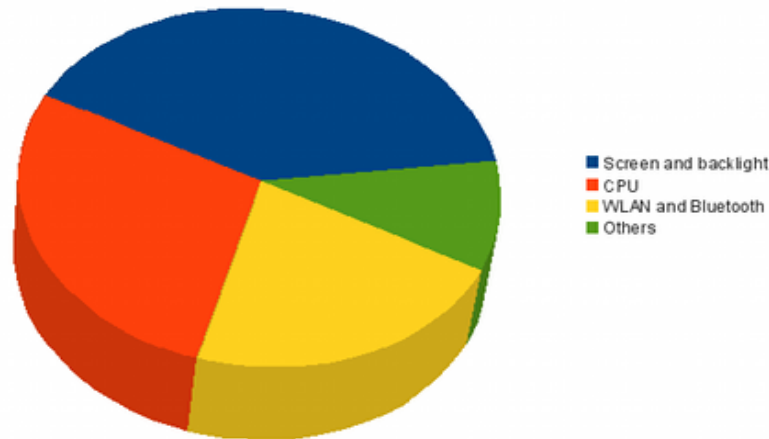
Maemo development environment provides many tools to monitor responsiveness and performance of applications [15].

Resource Use

Resource use has to be taken into account with maemo applications very carefully, because if an application takes up too much resources, it cannot be used. The resources that are scarce are RAM, flash, network bandwidth, file descriptors and IO bandwidth. Also besides base memory usage, also memory leaks must be naturally avoided. There are tools such as Valgrind[23] to help detecting possible memory leaks.

Power Consumption

Power Consumption in Mobile Devices



The device is not usable, if the user needs to recharge it all the time, or if it runs out of battery while idle. Therefore, no polling or busy looping should be used anywhere. Instead, an event-based approach, such as GTK's main loop is a better alternative. Also, too frequent timers or timeouts are to be avoided.

Secure Software Design

Quality and robustness together produce secure software. To achieve this it is important to follow secure software design guidelines and take into account certain issues specific to mobile Linux development.

1.1 Quality Awareness

This section is a generic test plan that defines a set of test cases for applications targeting the maemo platform. Good quality applications should pass all these test cases. If you see missing test cases in this section, please join "Quality Awareness" project in garage.maemo.org, and submit your cases to this material.

Test Cases:

1. *Packaging*

(a) Installation from application catalog

Precondition:

- Device without application and with default set of repositories.

Actions:

- Install application from install file (application catalog) by clicking install file in the browser.

Result:

- Application installs and runs correctly.

- All the needed dependences are downloaded automatically.
- There are no errors in the Application manager log.
- All icons created during installation are visible.

(b) Application upgrade

Precondition:

- Device with an older version of the application.

Actions:

- From the Application manager.
 - Update the repositories.
 - Upgrade application.

Result:

- The new application version works correctly in the device.
- There are no errors in the Application manager log.

(c) Application removal

Precondition:

- Device with the application installed.

Actions:

- Remove application by using Application manager.

Result:

- All the files installed by the package (except for modified configuration files) are removed.
- There are no errors in the Application manager log.
- Kill application process if it is running
- Remove tmp files

2. *Networking*

These cases assume that the application uses network.

(a) Offline mode

Precondition:

- Application is installed and works well in online mode.

Actions:

- Switch device to offline mode.
- Start application and try to use its networking functionality.

Result:

- Application should give indication to user that network is not available.

(b) Offline mode change

Precondition:

- Application is executing and works well in online mode.
- Working network available.

Actions:

- Start application.
- Switch device to offline mode.
- Try to use its networking functionality.

Result:

- Application should give indication to user that network is not available.
- If application UI blocks until network timeouts (in 30 seconds), it first indicates to user that it is working.

(c) Wlan disconnected

Precondition:

- Application is installed and works well in online mode.
- Working network available.

Actions:

- Disconnect network
- Start application and try to use networking functionality

Result:

- Device should connect automatically to network and application works correctly.

(d) WLAN and bluetooth connectivity works

Precondition:

- Application is installed and works well in online mode.
- Working network available and phone with data and bluetooth connectivity.

Actions:

- Run network test cases with WLAN connection
- Run network test cases with phone over bluetooth

Result:

- Application works with both connections

3. *Memory card*

Run these testcases on all memory cards.

(a) Memory card full

Precondition:

- Application installed and works.
- Fill the memory card.

Actions:

- Run the application

Result:

- Application should give correct error message

(b) No memory card

Precondition:

- Application installed and works.
- Remove all memory cards.

Actions:

- Run the application

Result:

- Application should give correct error message

(c) Memory card removed during application usage

Precondition:

- Application installed and works
- Memory card(s) inserted into slot

Actions:

- Start application
- Execute mmc functionality (e.g. access, save, delete, rename, backup)
- Remove mmc from slot

Result:

- Application should give correct error message

Observation:

- MMC functionality needs to take enough time (e.g. saving large sized file) so that memory card removing can be tested during its execution.

4. *Camera*

These cases are relevant if the application uses camera.

(a) Camera retracted

Precondition:

- Application installed and works
- Camera retracted

Actions:

- Start application
- Try to use camera functionality

Result:

- Application works and indicates that camera is not available

(b) Camera popped up during application usage

Precondition:

- Application installed and works
- Camera retracted

Actions:

- Start application
- Pop up camera
- Try to use camera functionality

Result:

- Camera functionality works correctly

(c) Simultaneous camera usage

Precondition:

- Application installed and works
- Other camera application running (e.g. Internet Call)
- Camera retracted

Actions:

- Start application
- Try to use camera functionality

Result:

- If application does not allow simultaneous camera usage, give the correct error message
- If application does allow simultaneous camera usage by either having its own switching mechanism or switching applications on the task navigator, camera functionality works correctly

5. *Platform Integration*

(a) Hildon UI style

Precondition:

- Application installed and works

Actions:

- Test the application so that all UI elements are visible

Result:

- Menus and toolbars are Hildonized
- Buttons sizes are usable with fingers
- Application icons are visible in menus & task bars
- HW Keys are mapped to relevant features
- Full screen mode works correctly

(b) Input method integration

Precondition:

- Application is installed to the device

Actions:

- Test all the text inputs and HW keys in the application UI

Result:

- Input method is invoked correctly when each of the text input widgets is tapped (by default normal input method for stylus tap and thumb keyboard for thumb tap) or if device has HW keyboard it is enabled and ready to use
- Enter key in the keyboard has logical behaviour
- HW keys have logical behaviour

(c) Backup

Precondition:

- Application is installed to the device

Actions:

- Change all application settings and quit the application
- Do a backup
- Flash the latest IT OS release
- Re-install the application
- Restore backup

Results:

- All changed application settings were restored for the selected categories

(d) Help

Result:

- Help item in application menu exists
- Help for the application in the device help exists

6. *Security*

(a) No unsecured TCP/IP ports

Precondition:

- Application is not installed
- Device has either ssh or terminal installed from which netstat can be run

Actions:

- Run "netstat -ta"
- Install application
- Run "netstat -ta"
- Run application
- Run "netstat -ta"
- Perform application use-cases and run "netstat -ta" after each use-case

Result:

- All identified ports should have documented security solution.

(b) Network source verification

Actions:

- Download code from or upload sensitive data to the network

Result:

- HTTPS or public key encryption used for the downloaded content

(c) Bluetooth security

Precondition:

- Application running in the device
- Ensure that your PC and tablet are not paired.

Actions:

- For each RFCOMM channel that your application opens, perform: "rfcomm connect <btaddr> <channel>". To find out which channel is used, publish it in SDP database and use "sdp-tool browse <btaddr>".

Result:

- Make sure that PIN is asked and user is asked to authorize connection in tablet before connection is established. Your security needs may vary, so authorization might not be necessary.

7. Performance

(a) Startup and close time

Actions:

- Boot the device
- Start the application (for the first time), and note when user can interact with the application
- Close application and note when application window disappears
- Start application again and note when the user can interact with the application

It is preferable to repeat these a few times and take the best times.

Result:

- Mandatory:

```
First startup < 6 secs
Second startup < 5 secs
Closing < 2 secs
```

- Desired:

```
First startup < 2 secs
Second startup < 1 sec
Closing < 0.5 secs
```

(b) UI responsiveness and feedback

Actions:

- Go through all application use cases

Result:

- All active UI elements react within 0.5 seconds to user actions (change state etc.)
- If an activity takes more than 3 seconds, there is progress indication

(c) Stress conditions

Precondition:

- "sp-stress" package installed

Actions:

- In (ssh) console run "cpuload 100"

Result:

- Application works without problems

8. *Battery usage*

Precondition:

- ssh package installed to the device and logged through ssh to the device

(a) No unnecessary CPU usage (no polling)

Precondition:

- Tested application is running, but it is not being used

Actions:

- Start "top" in (SSH) console
- Perform all application test cases

Result:

- After each application test case ends, all processes stop their CPU usage according to "top"

(b) No activity when device is idle or background

Precondition:

- "ltrace" package is installed to the device
- "No unnecessary CPU usage" test case is performed and application is left running

Actions:

- Wait until screen blanks
- ltrace the application process and if it's threaded, all of it's threads: "ltrace -S -p <PID1> -p <PID2> ..."
- Restart the application and repeat above steps

Result:

- Application has no activity when screen is blanked.

Note: threaded applications wake up at 8 sec intervals if they use LinuxThreads thread library instead of NPTL.

9. *Reliability*

(a) Memory leaks

Precondition:

- Application compiles and works in x86 Scratchbox
- Read Maemo Debugging Guide
- Install Valgrind to Scratchbox

Actions:

- Go through application use cases under Valgrind/Memcheck

Result:

- Valgrind does not report (definitive) memory leaks for the application

10. *Error handling*

(a) Full Flash memory

Actions:

- Fill internal device flash as "user" user, e.g. by copying a file in File manager
- Start the tested application
- Go through application use cases
- Exit application

Result:

- Application starts
- Application use cases that save data to Flash, give correct error message when the data save fails (this can mean that application shows an error at startup and exits after user OKs it)
- Application exits gracefully

Note: Part of the Flash is reserved for root, so applications cannot rely on just checking whether Flash has free space.

(b) Resource usage

Precondition:

- application running

Action:

- Start browser and browse to slashdot.org

Result:

- The application should not affect to browsing experience

11. *Scalability*

(a) Different amounts of input data

This concerns applications which can load external data, either from the filesystem or network.

Precondition:

- Start "top" in (SSH) console

Actions:

- Load data (files) with sizes of 1KB, 10KB, 100KB, 1MB, 10MB

With compressed formats, the data size can refer to uncompressed data size in memory.

Result:

- Loading of all files succeeds, at least when no other applications are running or swap is used
- Mandatory: The time application spends parsing/loading the data does not increase exponentially nor does the application memory usage increase exponentially

- Desired: With large data files, time and memory usage is less than linear in relation to the data size (application doesn't read all of the data, scales images on the run etc)

(b) Different screen sizes

Actions:

- Start application
- Switch application window between full screen and normal mode

Result:

- Both modes work

12. *Documentation*

Results:

- Release note
- Change log
- Copyrights in place
- Licensing in place
- Possibility to generate API docs from source if application provides APIs

13. *Policy conformance*

Results:

- Copyright & License defined
- Source code available if license require it
- Maemo trademark followed

1.2 Secure Software Design

1.2.1 Overview

This document tries to offer tools for secure software development, both with regard to processes and tools that can be used, and with special focus on mobile devices and the maemo platform. The target is to provide food for thought for anyone who is making a transition from non-mobile world to mobile world or closed-source to open-source, and would like to learn more about potential failure and vulnerability modes of mobile devices in general.

This document does not intend to be a definitive list of everything that could go wrong when developing software, nor does it try to describe how various attacks are specifically conducted. This sort of information is readily available both on-line and on flattened dead trees; please refer to the literature section for details.

The reader is expected to have knowledge about software development in general, including processes, implementation and testing.

Any tools or literature referred to in this document are listed only because it is believed that they represent potentially helpful resources. This document does not constitute an offer to sell or a solicitation to buy these goods or services.

1.2.2 Elements of Secure Software Development

Makings of secure software

Security in software can be defined in various ways, some addressing the functionality of the system and some building on the system's intended use. One of the best-suited definitions for general software development is a definition that is quality-based: software that fulfills its purpose can be considered as being of high quality. In a similar vein, software that fulfills its purpose under attack or in a hostile environment can be considered as being secure.

This definition already hints towards the main building blocks of secure software:

- The system designers need to understand the hostile environment in order to be able to defend against it; and
- The system needs to be tested properly in an environment that is a realistic hostile environment.

A quality-based definition also highlights synergies between quality and security as cross-cutting concerns. In the same way as a dedicated quality engineer cannot magically convert a product into a good quality one, a security engineer is also relying on everyone else's effort to actually make the product secure. And, as with quality, the metrics of security are not always straightforward and can often be measured only after the system has been taken into use.

Suggested Secure Software Development Lifecycle

Various different models for secure software development exist. One of the best documented is Microsoft's SDL (Secure Development Lifecycle), which has been documented in their book (see the literature section).

It is somewhat of a matter of taste, how many different steps should be included in a secure development lifecycle. However, when comparing the secure software development in many companies, three major steps can be seen as being common over the whole industry:

- Threat analysis
- Security-aware implementation
- Security testing

Of these, threat analysis and security testing are discussed below from the point of view of a mobile device. The second one, security-aware implementation, is obviously important as well, but learning about it is best left for existing literature.

On GNU/Linux (and other UNIX) platforms, perhaps the best source of ideas for security-aware implementation is Secure Programming on Linux and Unix HOWTO (again, see the literature section). There exists also a myriad of other books on this subject, and a proposed reading list can be found in the literature section of this document.

1.2.3 Threat Analysis

Threat Analysis as Basis for Secure Software

Attacking and defending an information system is an exercise in asymmetric warfare. The attacker can usually almost freely choose where to attack, whereas the defender must defend on all fronts.

In addition, malicious thinking does not always come naturally to everyone. Law-abiding people, who have no security background do not instinctively think like a wrongdoer.

As a consequence of both of these traits, a system that has been designed without specific security awareness may be open to attack from various angles. In order to counter this, one needs to conduct threat analysis, which is the basis for any security engineering work.

Threat analysis has two main objects. First, it tries to address the asymmetrical nature of security by identifying as many defensible assets and system aspects as possible. Second, the way in which threat analysis is conducted is designed to help the participants to think like an attacker.

How to Conduct Threat Analysis

There are many ways to do threat analysis. One of the more formal ones is to chart all data flows through an application by using an architectural diagram that displays all interfaces and components. This is best done when the system architecture is clear, and it is based on the observation that generally, most current attacks are due to malicious (malformed or somehow illegal) inputs, and software that fails when subjected to such input. By observing the movement of data through the various components of the application using an architectural diagram, sensitive interfaces can be identified and the main points of the system that need to be hardened can be picked out.

This type of data flow analysis does not, however, typically address cases where the system is attacked by other means than malicious inputs, for example by subjecting it to a different environment than where it was designed to operate. Hence, it is worthwhile also to use freeform methods, such as brainstorming in a group, to cover all possible ways in which an attacker might want to attack the system.

Brainstorming is also a good method to induce malicious thinking. Several people asking "but what if...", supporting each other and inventing new attack vectors can typically find much more creative ways of attacking a system than just someone who has no previous security experience. A hardened security practitioner may be able to pull off a successful security analysis alone, but for laymen (in terms of security), support from a group has been shown to enhance the quality of threat analysis.

Open Source Threat Landscape

General belief holds that open source practitioners are typically rather well security-aware. However, for someone coming from proprietary or closed-source software development, it might be helpful to discuss the specific traits that open source development brings on the security area. It should be noted

that these traits are not security threats by themselves, but they in many cases do have an effect on how security issues can be treated.

First, some open source licences require that any changes that have been made to source code need to be released under a similar licence. When considering security fixes, this means that fixes will get public, and usually it is very straightforward to compare a fixed version against an old version to determine what the security issue was. When utilising open source code, security issues are then automatically exposed in source code releases.

Second, many (but definitely not all!) open source projects are hobbyist projects, and in those cases the main author or maintainer of the project might not get paid for their work. This, in turn, may effect the turnaround time for security fixes, because some people may need to prioritize their other life (work and other personal interests) over the open source project. If your product is depending on a specific service level for security fixes, and the open source project is not one of the large established ones, you should probably be ready to resource this type of work yourself. This is also implied by typical open source licences, which usually release the code 'as is', without a warranty of any kind.

Third, open source may make it easier to develop a working exploit for a found security vulnerability. An exploit for a closed source code may be more of a hit-and-miss, but in open source the context of the vulnerability is clearly seen. This does not make open source any more insecure, but it may have an effect on the time that it takes to productize a security exploit.

On the other hand, open source has clear benefits for security: those open source projects that are widely used also tend to be kept looked at by many security researchers, so problems may get found quicker than in closed source. In addition, open source enables you to actually fix the problems yourself, if needed, so you can get on with the development of your system without having to wait for fixes to be included in closed source components.

Security in Mobile Devices

Mobile devices differ by nature from desktops or servers. Obviously the distinction between mobile and non-mobile systems are blurring all the time, and some of the considerations mentioned here will probably sound archaic in 2015. However, there are certain factors that will still differentiate between mobile and non-mobile devices for years to come, and therefore should be taken into account when thinking about security threats.

Three main categories of mobility-related threats are discussed shortly below:

1. Threats related to mobile data bearers
2. Threats related to the constrained system
3. Threats related to the personal nature of the device

First, the communication channel of a mobile device is typically much less wide than for a server or a desktop system. Even though 3G networks' HSPA, WiMAX and Wi-Fi networks are promising transmission speeds in several megabits per second, there may still be no guarantee that the channels will always be this fast. In rural areas with a 2G, non-EDGE-enabled GSM network,

the data transfer speed may be closer to a modem of early 1990s. From the perspective of an attacker, narrowband channels are easier to congest using a denial of service attack.

Another aspect which is related to the communication channels is the billing model. Most mobile devices insulate the running applications from the specifics of the data bearer. Data transfer might happen over Wi-Fi or cellular, and the bearer might even change during the operation of the application (for example, subsequent HTTP requests might be transferred over different bearers). Therefore, the applications cannot really make a guess whether the bearer they are using is flatrate or whether it is metered per-bit or per-second. Some cellular operators even meter packet data on a per-second basis, and some typically flatrate bearers such as Wi-Fi might have a per-bit charging model, which is still the case in certain hotspots such as hotels. If the application could somehow be coerced into producing a lot of traffic (either bitwise or timewise - think about background polling), this may result in a dramatic effect on the user's bills. Further, as an interesting twist, many operators operate premium-rate numbers for text messages and phone calls, and messages or calls to these numbers might cause unwanted costs - much in the same way as the 'modem hijacking' attacks were done when modem connections were the norm for home computers.

As the bearer might change dynamically, this also poses challenges to any network services that are being used. Cellular connections and text messaging are often thought as being secure, as they utilize closed networks that are typically well secured by operators. However, the bearer might suddenly change into an open Wi-Fi network, where all packets are open to casual inspection and any open ports that are listening to incoming data can be freely accessed. The connection security would really need to be designed with the least secure network in mind.

The second major difference between mobile and non-mobile devices is the power source. Battery technology has been historically very slow to develop, when compared to other features of portable devices. Power-saving and performance tuning of mobile devices is a very complex area, and a single ill-behaving program might cause excess power consumption. Even though this might not immediately sound like a security issue, it might turn out to be one if a flat battery causes a denial of service for the whole device.

This category of mobility-related challenges also applies to other resources than power. Mobile device CPUs usually run on smaller clock speeds, and they have less memory, less mass storage and a smaller screen than desktop or server machines. An attack against any of these more constrained resources might cause a denial of service.

A third category is concerned with the fact that mobile devices are often very personal. They share the same geographical location with their owner or user, and therefore might introduce a way for malicious software to track the user's whereabouts. Also, users typically store a lot of private data on the devices, such as address books, calendar markings, personal notes, photos, and of course, e-mail and text messages. Leaking this data may constitute a privacy problem, and deletion of the data may cause significant annoyance, as can be witnessed by anyone that has lost their mobile phone and has not made a back-up of the address book.

The personal nature of mobile devices is also highlighted in the way that unique but static identifiers are used. A typical mobile device has one or more

MAC addresses (for wireless LAN, Bluetooth, WiMAX), an IMEI (GSM/3G device serial number) and an IMSI (GSM/3G subscriber identity). As these identifiers are globally unique and static, combining them with the user may tag the user by a traceable identity, giving rise to various privacy threat scenarios. Software should therefore pay attention to privacy related issues and restrict the use of such identifiers.

1.2.4 Security Testing

Robustness Testing

So, a project did their security threat analysis and paid attention to secure implementation. The code is now ready - what happens next?

Typically, the implementation will be tested. Part of the testing is 'positive' testing: the implementation is subjected to use cases that are supposed to succeed. Some of the testing may be 'negative', in which the implementation is expected to fail cleanly.

Perhaps the most important security testing that should be performed is a specific type of negative testing. It is often called 'robustness testing', as it tries to make the implementation to break down, i.e. behave in a way that is not very robust. Looking back at the definition of security, this is a necessary part of a quality system being also secure: in a malicious or hostile environment (which is approximated by robustness testing), the system must still remain usable (meaning that it has to be robust).

Robustness testing usually concentrates on inputs to the system. Any system that interacts with its environment has some input interfaces. Typically they are protocol parsers, file parsers, user interface input elements, application programming interfaces, remote procedure call interfaces, and so on. The interface does not need to be intended to be used by a user - even a Domain Name System lookup, which returns something from a name server, is an interface, even though it is usually machine-to-machine. Therefore, any incoming data should be suspect and optimally should be robustness tested.

Obviously, testing everything may not always be an option due to lack of resources. Priority should be then given to those inputs that are unauthenticated (meaning that there is no way of telling from whom the input came) and those that get inputs from outside the system. As an example, robustness properties of a system API are probably tested with a smaller priority than a file parser that the user activates to read an input file, and again that file parser is probably lesser priority than a network daemon that listens on a network port and processes requests coming from an unknown remote computer. The security threat analysis should have already listed these interfaces and specified to which extent these would need to be robustness tested.

Robustness testing is typically built on inputs that are almost, but not quite, well-formed. Different robustness testing methods include fuzzing, in which a valid input is taken and (semi)randomly broken ("fuzzed"). Fuzzing can either happen on bit-level, especially when the input is binary data, or on a higher level that takes the input structure into account. As an example, an XML fuzzer might add non-specified XML elements or truncate existing XML elements.

Another, perhaps a more systematic way of creating robustness test cases is to synthesise broken test cases from the specification of the input. As an

example, if a specification says that a text field can be 16 bytes in length, test cases might try lengths of zero and 17 bytes. This sort of synthesis is typically more efficient than semirandom fuzzing, but comes initially with a much higher amount of work, and requires good knowledge of how parsers typically fail.

Bugs that are found with robustness testing often cause crashes or jams, or at least downgrade the usability. The target should be that any illegally formed content would cause minimum disturbance to normal use; in many cases, a silent discard of malformed data is the best approach to take, and sometimes the user might be notified with an understandable error message.

It may be useful to note at this stage that conformance test suites that exist for various protocols are not necessarily good security test suites. Conformance is by definition usually positive testing, and even though they may test failure modes, it is not guaranteed that these tools actually are malicious enough.

Pointers to robustness testing tools can be found in the literature section.

Static Analysis

Static analysis is a method where the source code itself is analysed to find potential security issues. Analysis tools differ in the ways they work, but typically they detect the use of dangerous functions and system calls, or track the flow of data through the components.

Static analysis tools may produce large amounts of false positives, and usually need to be taught to ignore some of the issues that they spot. Some more elementary tools may not be able to tell the difference between safe and unsafe ways of calling a potentially hazardous function, for example, whereas some sophisticated tools may actually track all the possible inputs to the function and evaluate security based on the input.

The best uses for static analysis are as a help for code review. Problems that are pinpointed by static analysis tools can be treated as a matter of course and code analysis effort can be then used for other types of problems. Large numbers of static analysis findings can also be used as an indicator that a specific part of code should be manually reviewed.

If the number of false positives has been driven successfully down, a static analysis system can also be utilised as a gatekeeper, when checking in new or changed code to version control system, or as one of the phases in unit testing. The first step in static analysis should probably be to turn on all compiler warnings and address all of those.

Some pointers to static analysis tools can be found in the tools and resources section.

1.2.5 Conflicts of Interest?

Usability and Security

Sometimes, usability is seen to be at odds with security, as security features may be perceived to be something that hinders the free use of the system. Even if this is true to some extent - for example, the concept of certificates is not a very easy one for the user to grasp - one could argue that this is actually more due to poor usability of security-related features than security itself. As an example, certificates could in most cases be totally invisible to the user, and new ideas

(a good example is Petname, see [18]) could be fielded that would change the whole security perception.

It is useful to understand that mobile devices are used by people who do not use a "normal" computer in everyday life. This trend will probably be even more significant in the future. Already now there exists a huge population, whose primary means of electronic communication is a mobile device, not a desktop, and they do not necessarily have the same model of (for example) web browser security in their mind as those who are accustomed to use a desktop browser.

When mobile devices are concerned, usable security engineering should be given a lot of thought. Most of the aspects are related to the restricted user interface. As an example, a typical secure password contains both uppercase and lowercase letters, and perhaps special characters. Typing such passwords on a typical mobile device keypad (such as the ITU-T numeric keypad) is time-consuming and error-prone. Therefore, a system that would do away with the need to enter passwords would be probably welcomed by users.

Another point is that all kinds of security indicators such as coloured URL bars, lock icons, and such, take up valuable screen estate.

Interoperability and Security

Interoperability and security may also be sometimes seen as being in conflict. To achieve maximum interoperability, a system "should be liberal in what they accept, and conservative in what they send". However, with liberalism comes a lot of responsibility: even though a system is liberal in accepting inputs, it should really know what it is getting before actually acting on the data.

1.2.6 Tools and Resources

Literature

For implementors on Linux and other Unix platforms, David A. Wheeler's Secure Programming on Linux and Unix HOWTO [25] is a good read. This document lists a lot of pitfalls on a Unix system. However, it does not really get in-depth with how the security is attained in a software project, but as an awareness-raiser for implementors, this is very useful (and free).

If you are managing a software development team, and are looking for insight as to how security could be brought into your software development methodology, SDL: The Security Development Lifecycle by Michael Howard and Steve Lipner (Microsoft Press, 2006) is a revealing account of how the processes were changed at Microsoft. The processes sound pretty waterfallish, and agile methods are given a somewhat cursory treatment, but this is still an interesting insight to how secure development can be pulled off in a large company. Microsoft's SDL people also blog about these issues at <http://blogs.msdn.com/sdl/>, where you can find pointers to SDL documentation updates.

Another book on how to implement security in software development is Secure Coding: Principles and Practices by Mark G. Graff and Kenneth R. van Wyk (O'Reilly, 2003). This is a very concise book, but somewhat misleadingly named - it does not address the actual implementation issues that much, but

instead lists tons of good principles in a checklist-style format for both architectural design and general software development.

Security and usability is not covered very well in books. However, there are two online resources that should be mentioned: the first is Ka-Ping Yee's User Interaction Design for Secure Systems [12], and the second is Peter Gutmann's manuscript Security Usability Fundamentals [19].

There are a few books on security testing, but none of them are above the rest. If you have a specific application in mind, browsing the contents of the security testing related books is recommended: some of them, for example, only talk about web application testing and therefore do not really lend themselves for learning about security testing generally.

Tools

Please note that these pointers are provided for informational purposes only. This document does not constitute an offer to sell or a solicitation to buy these goods or services.

There are a lot of fuzzers and it is not very difficult to implement one yourself, if your project calls for knowledge of the underlying data formats. A lot of tools can be found by searching the web or from Wikipedia [9]. Some of the freely available fuzzing tools are:

Untidy an XML fuzzer with a Python API [22].

WebScarab a fuzzer (and a lot more) for web applications [24].

zzuf a general-purpose binary fuzzer [27].

Bunny the Fuzzer a general-purpose, adaptive fuzzer that uses code instrumentation [1].

An example of a more systematic way for robustness testing are commercial tools developed by Codenomicon [2]. In their toolkits, problems are not randomly introduced but instead the test cases are built on the protocol or file format specifications.

Static analysis tools can also be found in both commercial and noncommercial domains. Commercial ones include offerings from Coverity [3], Fortify [8] and Klocwork [11], to name a few. Free ones include:

Flawfinder C and C++ source code scanner [7].

RATS C, C++, Perl, PHP and Python scanner [21].

For more options, please see Wikipedia [26].

1.2.7 Maemo Security Policies

Reporting Vulnerabilities

If you discover or become aware of a security issue in maemo software (consisting of the maemo platform software and the upstream projects whose software is used within the maemo platform software), please report it by email to security@maemo.org. Please encrypt your message with GnuPG using key ID

0x83AAAB3B (available from PGP key servers). The report will be analysed, and appropriate actions initiated.

If you discover a security issue in an upstream project whose code is used in maemo, or a 3rd party open source software running on the maemo platform, as your first priority, report the problem to the upstream project or their security team and only after that send a copy to maemo security as per above. If you do not know where to report the issue, we suggest you report it to the Open Source CERT at <http://www.ocert.org/>. Please see below for helpful tips on what information would be useful.

Please note that security@maemo.org only handles issues relating to the maemo platform. This email address does not handle security issues related to web sites (including the maemo.org website), 3rd party software running on the maemo platform, or issues specific to Nokia products. In any security issues related to these, please contact the appropriate party.

When reporting a security issue, it would be helpful to know if the security issue has been publicized somewhere, so if you can, please provide a pointer to that (web address, CVE identifier, etc.)

It is also important to understand, where and how the issue manifests itself. Information that is useful is the name of the affected component and package (and the full version number), configuration and environment where the issue was discovered (proof-of-concept code if available).

If you are able to provide more information and details that would be helpful in validating the issue, we would appreciate your contact information.

Any security-related bugs in maemo bugzilla should be tagged with keyword "security".

1.3 Maemo Coding Style and Programming Guidelines

The purpose of this section is to set the guidelines and recommendations for maemo developers, and anyone who writes new code for the maemo platform. This section assumes that programming is done in C language, but most of the guidelines are not language-specific. It is also required that developers possess a certain level of knowledge of Linux and GTK+ programming.

Why are programming guidelines needed? Some reasons for this are:

- Readability of code

Many people write program code for maemo and have different coding styles, because C allows it; therefore C programs can look very different, and be difficult to read for someone who is used to a particular style.

- Correctness and robustness of code

Some people are more disciplined programmers and others are more relaxed; this can lead to "sloppy programming" - that is, incomplete error handling, no sanity checks, hacking etc.

- Maintainability of code

Many programmers tend to write "ingenious" code which is almost impossible for others to read, and thus almost impossible to maintain and debug. So, things should be kept simple.

"Debugging is twice as hard as writing code in the first place. Therefore, if you write code as cleverly as possible, you are, by definition, not smart enough to debug it." (Brian W. Kernighan)

- Platform requirements

Some guidelines must be followed by maemo developers, because they have direct impact on the platform (for example, power usage).

In addition to this section, one should read also the GNOME Programming Guidelines [20] and pay attention to Glib/Gnome-specific issues. There is also a coding style document by Linus Torvalds for the Linux kernel [13], containing hints on how to configure one's editor with the specific indentation style used in the maemo code.

Even though proper coding style alone is not a guarantee of good software, it at least guarantees a good level of legibility and maintainability.

1.3.1 Maemo Coding Practices

Use the following maemo coding practices:

- The section below contains generic guidelines about coding style, variable usage and naming, error checking and many generic guidelines that good developers may already be familiar with. The most important thing about code style is consistency; one should pick a specific style and always follow it.
- Maemo developers must be careful with memory usage and memory leaks: one should always consider employing and initializing local variables. Code can be protected from memory leaks by freeing memory as soon as possible, or using automatic variables stored in a stack (not in a heap), guaranteeing automatic allocation and de-allocation.
- Memory pool should not be fragmented with `malloc(3)` and small or multiple instances of `free(3)`. Reusing memory or freeing it in large blocks is recommended. A more effective method than manually searching for memory leaks is to use an automatic tool. For more information, see section below.
- If planning to use threads, one should remember that thread based applications are more difficult to trace and debug. Threads should not be used just for their easiness or fashionability. They should be used only if they bring some clear benefits like better UI responsiveness to the application. Notice also that asynchronous calls to the functions can in some cases be used as replacements to the threads.
- System message notification and application-to-application communication make use of D-BUS, a lightweight message passing protocol. D-BUS must not be used to pass long or frequent messages between applications. D-BUS delivery is not guaranteed, but the application must check for it. For more information on D-BUS guidelines, see section below.

- When developing a new application in the maemo platform, it is the programmer's responsibility to catch hardware events, such as button press and touchpad actions, in the application. Touchpad events are similar to mouse events in a desktop environment. Hardware buttons form a special case, since they have special semantics, such as zooming, canceling operations and various other actions expected by maemo users. For more information on hardware events, see section below.
- Embedded devices offer many challenges, primarily for desktop developers. While a desktop user accepts a certain level of resource wasting, users of maemo devices do not accept such waste. One of the main objectives of maemo developers is to keep the resource usage at minimum. For a full review of this topic, see section below.

1.3.2 Basics of Good Programming

Most of the following conventions are useful for any programmer, and experienced programmers should already know them. However, there are also some that are related to programming for embedded devices, such as avoiding the use of dynamic memory, prioritizing the reliability of code and, above all, saving resources.

Coding Style

It is more preferable to use simple than "clever" solutions, and one should always be wary of lazy coding and especially so-called hacks. Clever code is harder to read, thus harder to debug and more difficult to prove correct. Careful coding means taking care of all possible error situations and taking the time to think while writing the code.

Source code should be commented. The comments should be written in English. In general, comments describe *what* is being done, not *how*. Comments should be made based primarily on what functions do, and they should always be kept synchronized with the code. The formats (such as strings, files and input) should be described to make it easier to understand the code.

It is not advisable to use more than three or four indentation levels. When using tab characters for indentation, one should make sure that the code does not overflow from a terminal screen (width 80 columns) even if the tab size is 8. The functions should be kept short, no longer than 50-100 lines of code, unless absolutely necessary. The line length should be kept at between 1-75 characters.

Empty lines should be used to group code lines that logically belong together. One or two empty lines can be used to separate functions from each other. The command "indent -kr <inputfile> -o <outputfile>" can carry out some of these functions for the code.

Variable Use

Local variables should always be favored over global variables, i.e. the visibility of a variable should be kept at minimum. Global variables should not be used to pass values between functions: this can be performed with arguments. Global

variables require extra care when programming with threads. Thus, these should be avoided.

All variables should be initialized to a known illegal value (such as 0xDEADBEEF), if they cannot be given a meaningful value from the beginning. There are at least two reasons for this:

- Known initial values contribute to predictable behavior in the program, i.e. they reduce random behavior in error situations caused by uninitialized memory. This helps in troubleshooting.
- Illegal initial values increase the probability of early detection of an erroneous state of the program, as an illegal value (such as a null pointer) is more easily detected than a random one.

Error Checking and Handling

The return codes of all system calls and functions that can potentially fail should be examined. Usually system calls only fail in rare situations, but these must also be dealt with. At least, it is good to have a log entry stating the reason for a program exit when the program cannot continue, instead of calling a segmentation fault when memory allocation has failed.

One should be prepared for network disconnections. If a program uses network connections, it must be written in a way that allows it to recover from lost connections and broken sockets. The program must operate correctly even when a disconnection occurs.

Variable and Function Naming

Descriptive, lowercase names should be used for variables and functions. Underscores (`_`) can be used to indicate word boundaries. Non-static and unnecessary global names should be avoided. If it is absolutely necessary to employ a global variable, type or function, one should use a prefix specific to the library or module where the name is, e.g. `gtk_widget_`.

Each variable should be employed for one purpose only. The optimization should be left to the compiler. Short names for local variables are fine when their use is simple, e.g. `i`, `j` and `k` for index variables. In functions with pointer arguments, `const` should be used when possible, in order to indicate what causes changes (or does not cause any changes) in the pointed memory.

Constants and Header Files

So-called magic values should not be used, meaning literal constants (such as numbers) should not be used as coefficients for buffer sizes. In a header file, `#define` can be used to name them. That name can then be used in the code instead of the value. That way, it is easier to adjust constants later and make sure that all occurrences are changed when the constant is modified. However, names like `#define ONE 1` should be avoided. If there are a lot of constants, putting them into a separate header file should be considered.

Header files should be written to be protected against multi-inclusion. Also, it should be made sure that they work when included from C++ code; this

is achieved by using `#ifdef __cplusplus extern "C" { ... }`, or by using `G_BEGIN_DECLS ... G_END_DECLS` in Glib/GTK+ code.

Memory Use

Static memory (variables) should be preferred to dynamic memory, when the needed memory size is not large and/or is used regularly in the program. Static memory does not leak, a pointer to static memory is less likely to cause a segmentation fault, and using static memory keeps the memory usage more constant.

Unneeded dynamic memory should always be freed in order to save resources. N.B. Shared memory is not released automatically on program exit.

Memory heap fragmentation caused by mixing calls of `free()` and `malloc()` should be avoided. In a case where several lumps of allocated memory have to be freed and reallocated, the use of `free()` must be a single sequence, and the successive allocation another sequence.

Thread Use

Threads can cause very nasty synchronization and memory reference bugs so you need to consider are advantages of using them (like improved UI responsiveness) bigger than disadvantages like more complicated and error prone design of your application.

Threads can be used especially for operations which may block application UI for several seconds and where canceling the operation (using the cancel dialog) is not possible. For example, network operations will block when the connection times out due to network becoming unreachable e.g. when user walks out of the Bluetooth or WLAN range.

Doing given functionality concurrently to the rest of the application requires that it is first well isolated from the rest of the code functionality-wise and that isolated code that interacts concurrently with rest of the application and platform uses only APIs which are thread safe. Only after making implementation thread safe code can be run safely in a separate thread.

Using threads may make communication with the other code easier, but as a downside it doesn't offer address space protection like process separation does, and it may be harder to debug and verify to work correctly in all situations.

1.3.3 Maemo Conventions

This section contains a collection of guidelines especially meant for maemo coding that complement those more generic rules given in previous sections.

D-BUS

The D-BUS message bus is a central part of the platform architecture. D-BUS is primarily meant for lightweight asynchronous messaging, not for heavy data transmission. This section contains conventions concerning the use of D-BUS.

One of the design principles of D-BUS was that it must be lightweight and fast. However, it is not as lightweight and fast as a Unix socket. By default, *D-BUS communication is not reliable*. The system has a single D-BUS system bus

that is used by many processes simultaneously. A heavy load on the system bus may slow down the whole system. To summarize:

- The D-BUS system bus should not be used for heavy messaging or regular transfers of large amounts of data, because it slows down the communication of other processes and requires extra processing power (and with it, electrical power). For that purpose, using a Unix socket, pipe, shared memory or file should be considered.
- Broadcasting on a D-BUS should be avoided, especially when there are a lot of listeners, such as on the system bus.
- When sending a message with, for example, `dbus_connection_send()`, it should not be assumed that the message is surely received by the server. Things can happen to make the message disappear on the way: the server can fail before receiving the message when the message is buffered in the client or in the daemon, or message buffer in the daemon or in the server can be full, resulting in discarding the message. Therefore, *replies must be sent and received in order to make sure that the messages have been reliably conveyed.*
- In D-BUS, messages specified by the application framework (including Task Navigator, Control Panel, Status Bar, and some LibOSSO messages), UTF-8 encoding should be used as the character encoding for all string values.
- The libOSSO library of the application framework contains high-level functions (`osso_rpc_*`) that wrap the D-BUS API for message sending and receiving. Notice that provided API for D-BUS is not thread safe and cannot thus be used from more than one thread simultaneously.
- All maemo applications need to be initialized with `osso_initialize()` function, connecting to the D-BUS session bus and the system bus. One symptom of missing initialization is that the application starts from Task Navigator, but closes automatically after a few seconds.
- The underscore (`foo_bar`) should be used instead of "fancy caps" (`FooBar`) in D-BUS names (such as methods and messages). The name of the service or the interface should not be repeated in the method and message names. The D-BUS method names should use a verb describing the function of the method when possible.

Applications should listen to D-BUS messages indicating the state of the device, such as "battery low" and "shutdown". When receiving these messages, the application may, for instance, ask the user to save any files that are open, or perform other similar actions to save the state of the application. There is also a specific system message, emitted when applications are required to close themselves.

Signal Handling

The application must `exit()` cleanly on the TERM signal, since this is the phase when GTK stores the clipboard contents to the clipboard manager, without losing them.

The TERM signal is sent, for example, when the desktop kills the application in the background (if the application has announced itself as killable).

Event Handling

The maemo user interface (Hildon) is based on GTK+ toolkit (the same as used in Gnome), so many aspects of maemo's event handling are similar to GTK+. There are a few points specific to maemo, since it is designed to be used with a touch screen. Some points to remember here are:

- It is not possible to move the cursor position in touch screen without pressing the button all the time. This difference may affect drawing applications or games, where the listening events need to be designed so that the user can click the left and right side of screen without ever moving with mouse from left to right.
- The touch screen can take only one kind of click, so only the left button of the mouse is used, leaving no functionality for the right button.
- When necessary to open menus using the right button, it is still possible to show them by keeping the button down for a little longer than a fast click. These kind of context-sensitive menus are supported by the platform. See the GTK+ widget tap and hold setup function in the Hildon API Documentation [14].
- GtkWidget in maemo has been changed to pass a special `insensitive_press` signal when the user presses on an insensitive widget. This was added because of actions such as showing a banner when pressing disabled widgets, e.g. a menu.

Touch Screen Events The main interaction with users in maemo is through touch screen events. The main actions are:

- Single tap
- Highlight and activate
- Stylus down and hold
- Drag and drop
- Panning
- Stylus down
- Stylus up
- Stylus down and cancel

Of course, all those touch screen actions are treated by GTK as regular mouse events, like button presses, motion and scrolling.

Minimizing Problems in X Applications The application must not grab the X server, pointer or keyboard, because the user would not be able to use the rest of the device. Exceptions: System UI application and Task Navigator, Status Bar, Application menu, context-sensitive menu and combo box pop-up widgets.

Applications must not carry out operations that may block, for example, checking for remote files to see whether to dim a menu item, while menus or pop-ups are open.

If the operation is allowed to block (for example, when the device gets out of WLAN or Bluetooth range), the whole device user interface freezes if blocking happens with the menu open. This kind of check must be performed before opening the menu or pop-up, or in a separate thread.

All application dialogs must be transient to one of the application windows, window group or other dialogs. Dialogs that are not transient to anything are interpreted as system modal dialogs and *block the rest of the UI*.

Hardware Button Events

Hardware key	Event
Home key	HILDON_HARDKEY_HOME
Menu key	HILDON_HARDKEY_MENU
Navigation keys	HILDON_HARDKEY_UP, HILDON_HARDKEY_DOWN, HILDON_HARDKEY_LEFT, HILDON_HARDKEY_RIGHT
Select key	HILDON_HARDKEY_SELECT
Cancel key	HILDON_HARDKEY_ESC
Full screen key	HILDON_HARDKEY_FULLSCREEN
Increase and decrease keys	HILDON_HARDKEY_INCREASE, HILDON_HARDKEY_DECREASE

It is important not to use the GDK_* key codes directly; the HILDON_HARDKEY_* macros should be used instead (they are merely macros for the GDK key codes). This is because hardware keys may relate to different GDK key codes in different maemo hardware.

Also, the application must not act on keystrokes unless it truly is necessary. For example, Home and Fullscreen keys have system-wide meanings, and the application can be notified by other means that it is going to be moved to background or put in full screen mode.

State Saving and Auto Saving

State saving is a method to save the GUI state of an application, so that the same GUI can be rebuilt after the application has restarted. In practice, the state is saved into a file stored in volatile memory, therefore the saved state will not last over a reboot of the device.

The application must save its state on any of the following events:

- When the application moves from the foreground to the background of the GUI.
- Before the application exits due to a memory management measure.

The application must always load the saved state (if it exists) on start-up. However, the application can decide not to use the loaded state, by its own calculation. The saved state must always be usable by the application, even if the application is likely to make the decision not to use it.

Applications must implement *auto saving* if they handle user data. User data is information that the user has entered into the application, usually for permanent storage, e.g. text in a note. Auto saving ensures that the user does not lose too much unsaved data, if the application or the system crashes, or the battery is removed. Auto saving can also mean that a user does not have to explicitly save their data.

Auto saving can be triggered by an event, e.g. the battery low event or system shutdown. The application must auto save:

- When it closes (for example, because of a system shutdown or restart).
- When it is moved to the background.
- At regular, configurable intervals when the application is on top, unless it has recently been auto-saved due to another event.
- On receiving a message requesting auto saving.

Changes made to dialogs are not auto saved. Every application must provide an event handler to implement the fourth item. Naturally, the application does not have to auto save when there is no new, unsaved data.

More information about state saving can be found in LibOSSO documentation [14].

Configurability

Maemo applications must use GConf for saving their configurations if configuration changes need to be monitored. GConf is the configuration management system used in Gnome. It is advisable to read the documentation available on the GConf project page [10].

Normal configurations (that do not need to be monitored) must be placed in a file under the `~/.osso/` folder. The file can be easily written or interpreted by the GKeyFile parser from Glib. Applications must have sensible defaults, ensuring that the application will function even if the configuration file is missing.

Using GConf

Naming GConf keys: first a new GConf directory (for GConf keys) should be made under the GConf directory `/apps/maemo` for the program. For example, if the program is named `tetris`, the private GConf keys go under the GConf directory `/apps/maemo/tetris`.

Notice that GConf operations do not have transactions so you need to be careful when updating or monitoring more than one configuration keys that have some dependency to each other.

File System

File management operations for user files must be performed through an API provided by the application framework. In other words: the Unix file I/O API should not be used for files and folders that are visible to the user. The Unix (POSIX) file I/O API can still be used in other parts of the file system.

The main reason for this is that maemo needs to make the file management operations behave case-insensitively, even on the case-sensitive Linux file system. Additionally, the auto naming of files and other such actions can be implemented in the application framework.

All applications are normally installed under the `/usr` directory, and must use the directory hierarchy described for `/usr` in Filesystem Hierarchy Standard (FHS) [6]. In addition to the directories specified in the document, the following are employed under `/usr`:

- `share/icons/hicolor/<size>/apps` for icon files
- `share/sounds` for sound files
- `share/themes` for GUI themes

Configuration files are placed under `/etc/Maemo/<program name>`.

Variable data files that are not user-specific (but system-specific) must be created under `/var`, as suggested in FHS.

User-specific files should preferably be located under the directory `/home/user/apps/<program name>`. The user's home directory `/home/user` can be used quite freely when there is no risk of a conflict. However, the directory `/home/user/MyDocs` is reserved for the user's own files (such as document, image, sound and video files), i.e. for files that are visible through the GUI.

The directory for the program temporary files is `/tmp/<program name>/`. To create a temporary file, `mkstemp(3)` immediately followed by `unlink(2)` is used. The reason for calling `unlink(2)` early is to ensure that the file is deleted even if the application crashes or is killed with the kill signal. Note that `unlink(2)` does not delete the file until the program exits or closes the file.

The available space on the main flash is very limited, and is very easily exhausted by applications or by the user (for example, by copying images, music or videos to it). If the flash is full, all the system calls trying to write to the disk fail, and the device is slower because JFFS2 file system needs some free space.

Therefore, no file can grow without limitation, and file sizes in general must be kept small. Applications must *not* write to flash when it is starting up, otherwise the write may fail and the application will not start.

Memory Profiling

Memory profiling is a difficult task and requires good tools. Great care must be taken considering memory leaks. Even though desktop users reboot the system often and memory leaks are less visible for them, maemo users expect to reboot the system less (or almost never), so even the smallest memory leak is noticeable.

There are many memory tools [15] available for profiling and finding memory leaks. Using Valgrind is recommended.

When compiling the program with debug and profile information, one should use standard profiling tools like `gprof` to find bottlenecks, as well as `valgrind` to find memory leaks or overruns. The `XResTop` tool (a top-like application to monitor X resource usage by clients) can also be used to monitor the leaking of X resources.

Secure Programming

This section lists conventions to be used to prevent security vulnerabilities in programs. At the same time, these can make programs more robust. For more information on secure programming, see [Secure Programming for Linux and Unix HOWTO \[25\]](#) and [Secure Programming \[16\]](#).

Hildon applications (such as GTK+) cannot run with elevated privileges, so that should be avoided in `suid/sgid` programs. If the applications really require higher privileges, the code must be split in two: a back-end should be created with higher privileges to communicate with the user interface through IPC.

These are the general rules for secure programming:

- Instead of `system(3)` and `popen(3)`, the `exec(3)` family of functions should be used when executing programs. Also, shell access should not be given to the user.
- Buffer overflows must be prevented.
- Backed-up data should be as secure as the original data, i.e. security and secrecy requirements for the back-up data must be the same as for the original data that was backed up.
- The default permissions for a new file (`umask` value) must have permissions for the user only.
- Temporary files must be created using `mkstemp(3)`.
- Storing secrets, such as passwords, directly into the code should be avoided.
- One should not use `crypt(3)`.
- One should not use higher privileges than necessary for the job at hand. Extra privileges should be dropped when they are no longer needed, if possible.
- The success of all system functions should always be checked, such as `malloc(3)`.
- Random numbers for cryptography must be read from the `/dev/random` or `/dev/urandom` device.

Buffer Overflows To avoid buffer overflows, safe versions of some system functions can be used. These can be seen in the table below. For Glib/GTK code, the Glib versions are best, e.g. `g_strlcpy()` guarantees that the result string is NULL-terminated even if the buffer is too small, while `strncpy()` does not.

System functions:

Unsafe	Safe	Best (GLib)
strcpy()	strncpy()	g_strlcpy()
strcat()	strncat()	g_strlcat()
sprintf()	snprintf()	g_snprintf()
gets()	fgets()	

When using `scanf(3)`, `sscanf(3)` or `fscanf(3)`, the amount of data to be read should be limited by specifying the maximum length of the data in the format string. When reading an environment variable with `getenv(3)`, no assumptions should be made of the size of the variable; it is easy to overflow a static buffer by copying the variable into it.

Packaging

All in maemo platform is installed as Debian packages. It is recommended to read more about packaging from Debian documentation [4], [5] and about maemo specific issues in packaging and package distributing from other parts of maemo documentation.

1.3.4 Power Saving

Modern CPUs are concerned about efficient power usage, and many techniques have been developed in order to reduce this power consumption. One of those techniques is CPU frequency reduction, which scales down the CPU's clock after a period of low usage.

Idle mode represents the CPU's state of minimum resource usage, and it is the best way available to save power. In this state, many core functions are disabled, and only when an external event happens, such as tapping on the screen, the CPU is awakened and starts consuming more energy.

The maemo developers must be aware that maemo platform employs these techniques to reduce power, and programs must be written so that they consume *as little electrical energy as possible* by means of giving maximum opportunity for the CPU to become idle.

Some useful tips:

- If the program is not processing or doing something useful, it should be left idle. Many toolkits (such as GTK+) have an `idle()` function that can be called when the program has nothing to do.
- The CPU should not be unnecessarily encumbered, as power consumption is proportional to the CPU load. Many processes running and competing for the CPU leads to more power waste.
- Periodical alarms or timers should not be used, unless absolutely necessary. One should consider what would be a reasonable period for updating a progress bar on the screen. Continuous (permanent) timers that tick faster than once in every five seconds must not be used.
- Busy-waiting mode is not acceptable, even if it is busy waiting on empty loops or NOPs: significant power saving is achieved only when the CPU

is idle. The following example shows how almost the same code can either prevent or support sleep.

Power Management Unfriendly code:

```
SDL_EventState (SDL_SYSWMEVENT, SDL_ENABLE);

while (!done)
{
    SDL_WaitEvent(0);
    while (SDL_PollEvent (&event))
    {
        if (event.type == SDL_SYSWMEVENT)
        {
            ....
            here, for example, check
            event.syswm.msg->event.xevent.type
            etc ...
        }
    }
}
```

Power Management Friendly code:

```
while(wait_outcome = SDL_WaitEvent(0))
{
    SDL_PollEvent (&event);
    if (event.type == SDL_SYSWMEVENT)
    {
        ....
        here for example check
        event.syswm.msg->event.xevent.type
        etc ...
        break; // <----NOTE THIS
    }
}
if(wait_outcome == 0)
    handle_sdl_error();
```

This optimization also provides power management friendly code with improved responsiveness, because it prevents a waiting application from monopolizing the CPU, leaving machine time to concurrent processes.

- All UI timers should be stopped when moving to the background, and when the screen is turned off. Application-specific exceptions for this rule can be granted, when necessary.
- Updating the GUI should be avoided when the application is running on the background, or when the screen has been turned off. Unnecessary graphical elements or constantly updated screen components should also be removed.
- Doing polling for a resource wastes CPU and power; instead, asynchronous notifications sent by a separate daemon should be used.
- Small writes to file system should be grouped into bigger chunks.
- The use of remote peripherals, such as Bluetooth or Wireless, should be minimized. It is advisable to keep values in cache and avoid repeated and/or redundant queries to such devices. The use of hardware in general should be minimized. This refers to e.g. screen updates or external storage devices.

For more information, see Software Matters for Power Consumption [17].

Bibliography

- [1] Bunny the Fuzzer. <http://code.google.com/p/bunny-the-fuzzer/>.
- [2] Codenomicon. <http://www.codenomicon.com>.
- [3] Coverity. <http://www.coverity.com/>.
- [4] Debian - The Universal Operating System. <http://www.debian.org/>.
- [5] Debian New Maintainers' Guide.
<http://www.debian.org/doc/manuals/maint-guide/>.
- [6] Filesystem Hierarchy Standard. <http://www.pathname.com/fhs/>.
- [7] Flawfinder. <http://www.dwheeler.com/flawfinder/>.
- [8] Fortify. <http://www.fortifysoftware.com/>.
- [9] Fuzz. http://en.wikipedia.org/wiki/Fuzz_testing.
- [10] GConf configuration system. <http://www.gnome.org/projects/gconf/>.
- [11] Klocwork. <http://www.klocwork.com>.
- [12] Ka-Ping Yee's User Interaction Design for Secure Systems.
<http://www.ischool.berkeley.edu/~ping/sid/uidss.pdf>.
- [13] Linux kernel coding style.
<http://pantransit.reptiles.org/prog/CodingStyle.html>.
- [14] Maemo API reference. http://maemo.org/api_refs/.
- [15] Maemo SDK tools. <http://maemo.org/development/tools/>.
- [16] Oliver Friedrichs (of@securityfocus.com). Secure Programming.
<http://marcin.owsiany.pl/sec/secure-programming.html>.
- [17] Nathan Tennes on Embedded.com. Software Matters for Power Consumption. <http://www.embedded.com/story/OEG20030121S0057>.
- [18] Petname. <http://petname.mozdev.org/>.
- [19] Peter Gutmann's manuscript Security Usability Fundamentals.
<http://www.cs.auckland.ac.nz/~pgut001/pubs/usability.pdf>.

- [20] Federico Mena Quintero, Miguel de Icaza, and Morten Welinder. GNOME Programming Guidelines. <http://developer.gnome.org/doc/guides/programming-guidelines/book1.html>.
- [21] RATS. <http://www.fortifysoftware.com/security-resources/rats.jsp>.
- [22] Untidy. <http://untidy.sourceforge.net/>.
- [23] Valgrind. <http://valgrind.org/>.
- [24] WebScarab. http://www.owasp.org/index.php/Category:OWASP_WebScarab_Project.
- [25] David A. Wheeler. Secure Programming for Linux and Unix HOWTO. <http://www.dwheeler.com/secure-programs/>.
- [26] Wikipedia List of Tools for Static Code Analysis. http://en.wikipedia.org/wiki/List_of_tools_for_static_code_analysis.
- [27] zzuf. <http://sam.zoy.org/zzuf/>.