



Primer on Reversing Symbian S60 Applications

Version 1.4

Last Rev.: June 2007

Into this Tutorial

1. Few worlds on what is Symbian S60
2. Instrument your Reversing Lab
3. Reversing the first application: SpriteBackup
4. Reversing the second application: CoolMMS
5. Reversing the third application: MIABO a MIDP 2.0 Java application
6. Some Protection Schemes
7. Conclusions
8. Further Readings

Author: Shub-Nigurath



Forewords

Time has come to write a new tutorial. This time I will leave the Win32/Windows world to open a new window (©) into another world.

Symbian phones, mostly Nokia, have started to become really interesting and powerful. Important applications appeared on the market and the system became a really general purpose operating system. The applications started to be protected with important tricks and some tools appeared which can handle them.

Unfortunately the Symbian scene is not so prolific of tutorials and what I found after a lot of searching and talking with others guys are just a few simple and quite old tutorials and few advanced things, mostly not written in English.

I decided then to take a long journey into this world, examining which tools you can use to disassemble the Symbian programs, how to approach to them and what generally you can do to create and distribute patches for those applications.

I started from the ground up, just because as said there were no discussion forum like our (at least I have not found them) where one can ask, the present special issue collects a series of single tutorials I wrote with different targets and difficulty levels. Probably the few Symbian groups around will laugh at me for the simple or even not correct approach, but as usual if one knows things better he should write a tutorial to demonstrate it.

The tutorial will cover different issues:

- Few words on the Symbian OS
- What instruments we have and what to use and customize them (particularly IDA)
- Practical examples of real applications

I also included a long list of references and further readings, as usual.

The commercial applications I used have been selected using two criteria:

- being educative for my purposes
- being already cracked by someone else, so as to not create problems on my own

Have phun,
Shub

Disclaimers

All code included with this tutorial is free to use and modify; we only ask that you mention where you found it. This tutorial is also free to distribute in its current unaltered form, with all the included supplements.

All the commercial programs used within this document have been used only for the purpose of demonstrating the theories and methods described. No distribution of patched applications has been done under any media or host. The applications used were most of the times already been patched, and cracked versions were available since a lot of time. ARTeam or the authors of the paper cannot be considered responsible damages the companies holding rights on those programs. The scope of this tutorial as well as any other ARTeam tutorial is of sharing knowledge and teaching how to patch applications, how to bypass protections and generally speaking how to improve the RCE art. We are not releasing any cracked application.

Verification

ARTeam.esfv can be opened in the ARTeamESFVChecker to verify all files have been released by ARTeam and are unaltered. The ARTeamESFVChecker can be obtained in the release section of the ARTeam site: <http://releases.accessroot.com>

Table of Contents

Forewords	1
1. Few words on what Symbian S60 is	4
1.1. Few details on the operative system	5
1.1.1 Phone Drives structure	6
1.1.2 The Developing Process and the applications structure	7
2. Instrument your Reversing Lab	8
2.1. How to install applications on the phone, PC Suite and Nokia 6600	8
2.2. Unpacking a SIS file	9
2.2.1 Existing tools to handle SIS files	10
2.2.1.1 Tools to handle SIS files	10
2.2.2 SIS File Structure	10
2.2.3 A Note on SISX Files	12
2.3. Other Useful Tools	12
2.4. Useful Tools working on the phone	14
2.5. Analysis done by IDA	15
2.5.1 For those completely unaware of what IDA is	15
2.5.2 Improving the IDA Analysis adding missing IDS	16
2.5.2.1 The EFD utility	18
2.5.3 Improving Imports readability: undIDC tool	18
2.5.3.1 Using the undIDC custom tool to simplify IDA Imports demangling	19
2.5.3.2 Alternative Syntax of undIDC	21
2.5.3.3 What undIDC does	22
2.5.4 Resolving Stubs used by the Compiler	23
2.5.5 Strings analysis of Symbian programs with IDA	23
2.5.6 Using desquirr decompiler	24
2.5.7 Reassume of the steps required to get your IDA dressed for Symbian	24
2.5.8 Reassume of the settings most suitable to get IDA dressed for Symbian	25
2.6. ARM Assembler	27
2.7. The Symbian Scene and Binary Diffing Suite	28
2.7.1 Symbian Scene?	28
3. Reversing the first application: SpriteBackup	29
3.1. How to crack this nut	30
3.1.1 Using desquirr to confirm the guess and find other ways to peel the cat	34
3.1.2 Patching the target and testing	35
3.1.3 Creating a new SIS	38
4. Reversing the second application: CoolMMS	39
4.1. How to Crack this nut	39
4.1.1 Creating the first patch	44
4.1.2 Trying out the first patch and find the other	45
4.1.3 Trying out the application with the two patches	47

4.2.	Creating the new SIS.....	47
4.3.	Using the EBDs suite.....	49
5.	Reversing the third application: MIABO a MIDP 2.0 Java application	52
5.1.	Approach used for the rest of this section	52
5.2.	How to Crack this nut.....	53
5.2.1	Finding the target on the phone and decompiling it on the PC	53
5.3.	Where to get JVM bytecode specifications.....	55
5.4.	Trying out a first patch	55
5.5.	Trying out a second patch	57
5.6.	Trying out a third patch	60
5.6.1	Analyzing the function q()	61
5.6.2	Creating the patch	63
5.6.3	Changing the About message.....	66
6.	Some misc Protection Schemes	69
6.1.	Using the IMEI and IMSI numbers for Registration Schemes.....	69
6.1.1	Approaching again SpriteBackup	69
6.1.2	Lesson learnt.....	72
6.2.	A simple protection using a complex license manager framework	73
6.2.1	Patch the application	74
6.2.2	Lesson learnt.....	76
6.3.	A Complex Protection with the classical heel of Achilles: ProfiMail 2.56	76
6.3.1	Packaging the new sis file.....	78
6.3.2	Lesson learnt.....	79
7.	Conclusions & Further Readings.....	79
8.	References	80
9.	Greetings.....	81
	Document History	81

1. Few words on what Symbian S60 is



I specifically concentrated on the Symbian OS v7.0 S60 operating system, almost because I have a Nokia 6600 model. Usually this version is called Symbian S60 (not to be confused with S60 3rd edition).

Unfortunately, by the reversing point of view, there are several versions of the Symbian OS and several series, most of which are not compatible at binary level, but all shares a common set of functionalities. Programs are distributed by producers for each phone model, usually you have to select the phone model and then the distributor's site will give you the correct version of the program.

This is IMHO one of the reasons why there are not much tutorials around on this OS. After all these phones are working with ARM processors which at assembler levels are all the same thing.

The fact that ARM is a RISC processor (Reduced Instruction Set CPU, so more difficult assembler) is IMHO another important obstacle.

Symbian is currently owned by Ericsson (15.6%), Nokia (47.9%), Panasonic (10.5%), Samsung (4.5%), Siemens AG (8.4%), and Sony Ericsson (13.1%).

Consider that even for just the Symbian versions 7.0 there are different Series to consider, which differs mainly for the users interfaces and services:

- S60, released in 2002, it's one of the most used versions, used for a lot of top-selling phones (Nokia 6600, 6630, N-gage and all the N series, like recent N-70) and absolutely the most popular and stable version.
- S80, used mostly by communicators (like Nokia 9210)
- S90, another version less spread especially developed for supporting special devices. Unfortunately, Series 90 is completely incompatible with Series 60.

The main CPUs of all these phones are ARM compatible chips.

Here I report using the Wikipedia page (http://en.wikipedia.org/wiki/Symbian_OS) the most relevant today Symbian releases.

- Symbian OS v7.0 and v7.0s. First shipped in 2003. This is an important Symbian release which appeared with all contemporary user interfaces including UIQ (Sony Ericsson P800, P900, P910, Motorola A925, A1000), Series 80 (Nokia 9300, 9500), Series 90 (Nokia 7710), Series 60 (Nokia 6600, 7310) as well as several FOMA phones in Japan.

In 2004, Psion sold its stake in Symbian.

Also in 2004, the first worm for mobile phones using Symbian OS, Cabir, was developed, which used Bluetooth to spread itself to nearby phones.

- Symbian OS v8.0. First shipped in 2004, one of its advantages would have been a choice of two different kernels (EKA1 or EKA2). However, the EKA2 kernel version did not ship until SymbianOS v8.1b. The kernels behave more or less identically from user-side, but are internally very different. EKA1 was chosen by some manufacturers to maintain compatibility with old device drivers, whilst EKA2 offered advantages such as a hard real-time capability. v8.0b was deproductized in 2003.
- Symbian OS v8.1. Basically a cleaned-up version of 8.0, this was available in 8.1a and 8.1b versions, with EKA1 and EKA2 kernels respectively. The 8.1b version, with EKA2's single-chip phone support but no additional security layer, was popular among Japanese phone companies desiring the real-time support but not allowing open application installation.
- Symbian OS v9.0. This version was used for internal Symbian purposes only. It was deproductized in 2004. v9.0 marked the end of the road for EKA1. v8.1a is the final EKA1 version of SymbianOS. Symbian OS has generally maintained reasonable binary compatibility. In theory the OS was BC from ER1-ER5, then from 6.0 to 8.1b. Substantial changes were needed for 9.0, related to tools and security, but this should be a one-off event. The move from requiring ARMv4 to requiring ARMv5 did not break backwards compatibility. Anyway Symbian developer proclaims that porting from Symbian 8.x to Symbian 9.x is a more daunting process than Symbian says.

- Symbian OS v9.1 released early 2005. It includes many new security related features, particularly a controversial platform security module facilitating mandatory code signing. Symbian argues that applications and content, and therefore a developers investment, are better protected than ever, however others contend that the requirement that every application be signed (and thus approved) violates the rights of the end-user, the owner of the phone, and limits the amount of free software available. The new ARM EABI binary model means developers need to retool and the security changes mean they may have to recompile. S60 3rd Edition phones have Symbian OS 9.1. Sony Ericsson is shipping the M600i based on Symbian OS 9.1 and should ship the P990 in Q3 2006. The earlier versions had a fatal defect where the phone hangs temporarily after the owner sent hundreds of SMSes :-O
- Symbian OS v9.2 released Q1 2006. Support for Bluetooth 2.0 (was 1.2) and OMA Device Management 1.2 (was 1.1.2). S60 3rd Edition Feature Pack 1 phones have Symbian OS 9.2.
- Symbian OS v9.3 released on 12 July 2006. Upgrades include native support for WiFi 802.11, HSDPA, Vietnamese language support.

On November 16, 2006, the 100 millionth smartphone running the OS was shipped.

1.1. Few details on the operative system

Symbian OS is an operating system, designed for mobile devices, with associated libraries, user interface frameworks and reference implementations of common tools, produced by Symbian Ltd. It is a descendant of Psion's EPOC and **runs exclusively on ARM processors**.

There are multiple platforms, based upon Symbian OS, which provide an SDK for application developers wishing to target a Symbian OS device - the main ones being UIQ, Series 60, etc. Individual phone products, or families, often have SDKs or SDK extensions downloadable from the manufacturer's website too. The SDKs contain documentation, the header files and library files required to build Symbian OS software, and a Windows-based emulator ("WINS"). Up until Symbian OS version 8, the SDKs also included a version of the GCC compiler (a cross-compiler) required to build software to work on the device.

Unfortunately WINS is an Emulator and not a Simulator. The difference is great: emulators emulate the real device but are indeed Win32 or PC programs which are built to look like real phones, simulators instead, using real device ROMs acts just like the real devices doing calls to the ROM of the device. This difference, as we will see doesn't allow simulating **any** phone application on a PC. You can only execute on the PC just those applications for which you have the sources and that you are able to compile: you must compile the application telling that the destination platform is WINS. This is a limitation by design and to do a real on-device debugging you can buy a tool from CodeWarrior¹ or buy a hardware card which allows remote debugging on the phone from the PC.

Symbian OS 9 requires a new compiler - a choice of compilers is available including a new version of GCC (the free C/C++ compiler of Linux, available also on Windows). In terms of SDKs, UIQ Technology now provides a simplified framework so that the single UIQ SDK forms the basis for developing on all UIQ 3 devices, such as the Sony Ericsson P990 and Sony Ericsson M600.

Symbian C++ programming is commonly done with a commercial IDE. For current versions of Symbian OS, CodeWarrior for Symbian OS is favored. The CodeWarrior tools will be replaced during 2006 by Carbide.c++, an Eclipse-based IDE developed by Nokia. It is expected that Carbide.c++ will be offered in different versions: a free version may allow users to prototype software on the emulator for the first time in a free product.

Visual Basic, VB.NET, and C# development for Symbian can be accomplished through AppForge Crossfire, a plugin for Microsoft Visual Studio (Appforge will be discussed into another specific tutorial).

There's also a version of a Borland IDE for Symbian OS. Symbian OS development is also possible on Linux and Mac OS X using tools and techniques developed by the community, partly enabled by Symbian releasing the source code for key tools.

Once developed, Symbian OS applications need to find a route to customers' mobile phones. They are packaged in SIS files which may be installed over-the-air, via PC connect or in some cases via Bluetooth or memory cards. An alternative is to partner with a phone manufacturer to have the software included on the

¹ http://seap.forum.nokia.com/info/sw.nokia.com/id/204ab18e-410c-4c59-bcd4-dda936c8a79b/CodeWarrior_On_Device_Debug_Kit_for_Series_60_3rd_Edition.html

phone itself. The SIS file route will be a little more difficult from Symbian OS 9, because any application wishing to have any capabilities beyond the bare minimum must be signed via the "Symbian Signed" program.

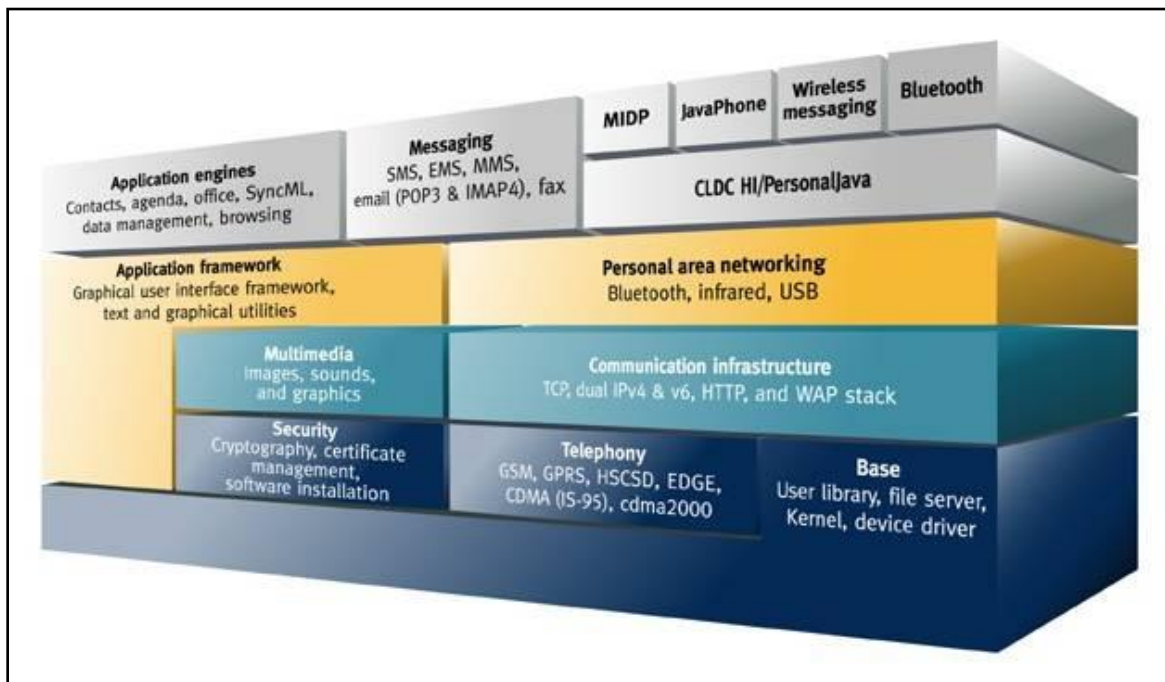


Figure 1 – Overview of SymbianOS Services

As shown in Figure 1 it's not only possible to develop using C/C++ or VB bridging frameworks, but also using Java. Java ME applications for Symbian OS are developed using standard techniques and tools such as the Sun Java Wireless Toolkit (formerly the J2ME Wireless Toolkit). They are packaged as JAR (and possibly JAD) files. Both CLDC and CDC applications can be created with NetBeans. SymbianOS also support the MIDP 2.0 Java specifications (a particular standard java profile/framework for mobile phones, common to most modern models), beside the java support. These specifications are meant to be vendor and model independent. Reversing and patching Java mobile applications is the simplest thing.

Nokia Series 60 phones can also run python scripts when the interpreter is installed, with a custom made API that allows for Bluetooth support and such. There is also an interactive console to allow the user to write python scripts directly from the phone.

1.1.1 Phone Drives structure

Few short things might become useful later on and are required to find data around [9].

Symbian file system is based on drive letters, directories and files

- C: FLASH RAM User data and user installed applications
- D: TEMP RAM Temporary file storage for applications
- E: MMC card Removable disk for pictures and applications
- Z: OS ROM Flash drive that contains most of the OS files

All drives have System directory:

- The directory is created automatically on a new media when one is inserted
- The System directory contains directory tree that contains OS and application files. Very much the same as C:\windows

Most important directories:

- System\Apps Applications that are visible to user
- System\Recogs Recognizer components
- System\Install Data needed for uninstallation of user installed applications
- System\libs System and third party libraries

Implementation Of User Services.

All phone features are implemented using .APP GUI applications. Anything that is visible in phone menu or started through buttons, is actually application under Z:\System\apps\

- Z:\System\Apps\Menu\Menu.app
 - Phone main menu and application launching service
- Z:\System\Apps\AppInst\Appinst.app
 - Application installation
- Z:\System\Apps\AppMgr\AppMgr.app
 - Application uninstallation
- Z:\System\Apps\MMM\Mmm.app
 - Messaging application for sending and receiving SMS,MMS,BT
- Z:\System\apps\phonebook\Phonebook.app
 - Phonebook
- Z:\System\apps\btui\btui.app
 - Bluetooth control panel

If any of the user service applications is disabled, user cannot use that feature anymore, till hard reset of the phone.

1.1.2 The Developing Process and the applications structure

The developing process of a Symbian application is a little different from the usual developing and building process of normal PCs applications. I found a very interesting schema, reported in Figure 2.: it reports the 3 main developing phases, building the application, building the resources and finally building the sis file for distribution. Later into this document the meaning of the files will be clearer. What is important now is underlining that Symbian applications are not only made of an executable (the .app file) but also of different files, first of all the resources file, which are really important for applications patching.

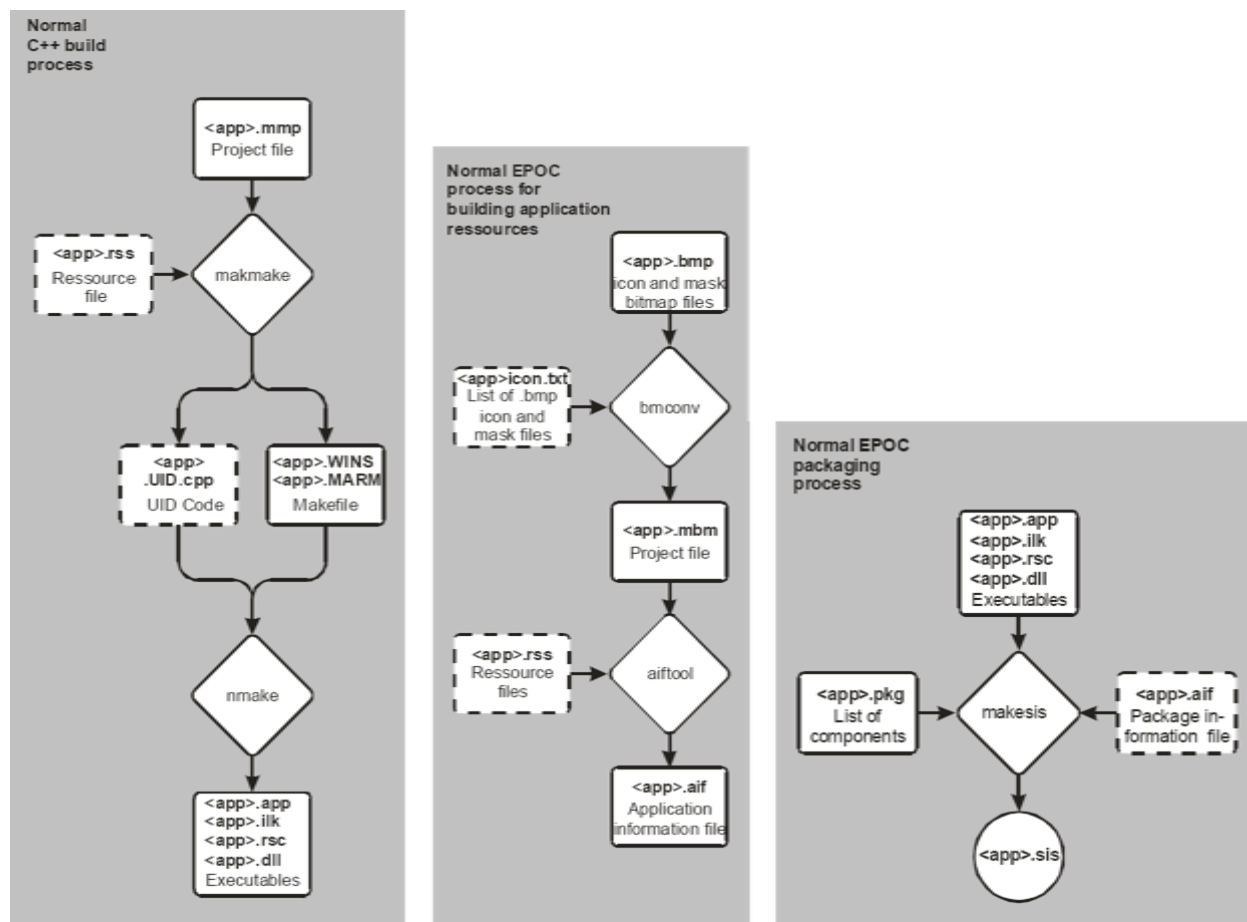


Figure 2 - C/C++ developing process

The resulting application will have a structure like the following one, see Figure 3. The Engine is usually completely separated from the User Interface. Keeping in mind this structure is really important for reversing applications because will help to find where to find the relevant code.

I suggest reading the document [43] for a quick introduction about these concepts, and the basic data structures you need to know in order to reverse applications.

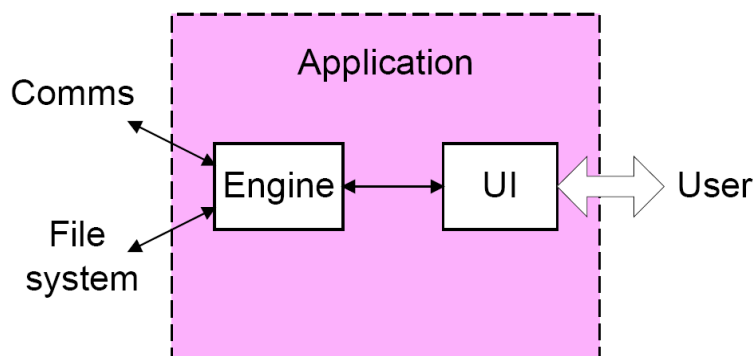


Figure 3 - General components of an Application

2. Instrument your Reversing Lab

Basically the problem with Symbian Phones (not only Nokia then), is that a real debugger doesn't exist, or better, it's not cheap and requires a proper hardware to fully debug applications on-device. This practically means, for us poor independent reversers that it doesn't exist. Fortunately IDA Interactive Pro supports Symbian (not version 9.x yet, anyway), then at least a dead-listing is possible.

The approach for Symbian reversing follows then always the following main steps:

1. decompile the distribution files (installation archive, .sis or .sisx file)
2. search the application file (*.app) and feed it to IDA
3. wait till it finishes
4. try finding the correct patch
5. change the bytecode, using the ARM opcodes tables (see later)
6. test the application uploading on the target phone, via Bluetooth for example
7. Repack a new .sis file to distribute your newly patched application

As you can see from the steps above there are some things which require tools: a SIS files extractor first of all, IDA of course, a SIS creator, a table of ARM opcodes, and few other things for example for exploring programs resources or convert special "mbm" bitmaps, used by Symbian, to/from normal bitmaps.

2.1. How to install applications on the phone, PC Suite and Nokia 6600

I added this section to the tutorial, even if it should be an already known issue, but it is just to underline some points. The Nokia phones communicate with PCs using the Nokia PC Suite, which is a program well documented in your phone manuals, so I will not argue on how to use it. Anyway I would pinpoint some things, especially for the Nokia 6600 phone. The 6600 requires a special PC Suite version, different by all the others. This PC Suite has not been updated since long time, so nowadays gives a lot of problems I would underline here.

First of all this version, is only Compatible with the following Bluetooth Connection Stacks:

- Digianswer Bluetooth Software Suite
- Microsoft Windows XP Bluetooth
- WIDCOMM Bluetooth for Windows version 1.4

If you don't have one of them you cannot use the PC Suite via Bluetooth. While the normal PC Suite 6.x for other phones also supports Toshiba Bluetooth for example, the version for 6600 doesn't.

This means your options are:

- Live without PC Suite

- Use infrared for PC Suite (assuming your Toshiba has that)
- Get an additional Bluetooth adapter supported by the 6600 version of PC Suite
- Get another notebook with such a Bluetooth adapter built-in
- Get another phone, which is supported by PC Suite 6.x
- Use the infrared connection.



Obviously I chosen the last point, but it requires a little of work. You have to disable the Connection Manager, the program with an icon like here beside.

To prevent the Connection Manager from launching on Windows startup, just do the following steps:

1. Open regedit
2. go in the tree on the registry editor's left side go to the key "HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows\CurrentVersion\Run".
3. Find the value "Nokia Tray Application". This might be something like "c:\Program Files\Common Files\Nokia\NCLTools\NclTray.exe".
4. Delete "Nokia Tray Application".

Disabling the Connection Manager allows to fully use the PC Suite over infrared and also some of the tools I will introduce in later sections²

2.2. Unpacking a SIS file

SIS files are the way to distribute EPOC applications (Symbian was previously known as EPOC and its still this way that IDA calls the Symbian applications). Not only do they install the software, but they also allow the application to be easily removed from the machine.

SIS files have a well documented structure which is handled by existing tools.

If you want there's an excellent description of the format available at [1]

Essentially SIS files (also the SISX format, introduced with Symbian S60 3rd Edition) can be seen as an archive with a fixed folder structure, used by the system to understand where to install the programs. What the phone OS requires is just where to place things and not much more, so installers are just copying files in the proper places, check if the OS is the correct one (to avoid installing on an unsupported phone) and add uninstallation information³. If you are able to open a SIS file you will find buried into a folders tree the real applications files, in the same order you will find into the phone.

The information in the .sis files is used by the system for three related, but different activities:

- Installing Applications
- Removing Applications
- Updating an existing Application

The uninstallation or upgrade information are stored into a sis file (with same name of the installed one), created into the phone /System/Install folder (on the flashcard or on the real phone, depending on where the application has been installed). Note that this is not the original file, it only contains information about the installed files, and is used on upgrades and removal. During removal, all the files written inside will be removed. Remove operation does no check if other applications are still using the same files or if some files are left in place, it just remove things listed in the special removing sis file⁴.

² For example the SymbFS for Total Commander won't work on IR 'till Connection Manager is removed

³ Indeed SIS files have also other things, such as applications certificates, versions, language supported, and so on, but most of these things, are often irrelevant for us.

⁴ Note that Symbian is a fully Unicode system, then all the strings are usually Unicode

2.2.1 Existing tools to handle SIS files

The Official Nokia Symbian SDK [2, 3] contains also the tools required to distribute your own applications through a SIS file, but the SDK is quite long to download and big to install. Fortunately the SIS management can be done even without the whole SDK.

2.2.1.1 Tools to handle SIS files

Tools to Extract SIS files

The ones I use most are:

- *SisView* [4] is a freeware plug-in created for the shareware program Total Commander. This tool allows you to view the contents of any .sis file just like any other normal archive.
 - *SYMBFS - Symbian Filesystem plugin for Total Commander* [5], which allows direct browsing from Total Commander the file system of a Nokia phone (connected with BlueTooth Serial Port Profile). This gives great freedom searching the right programs and uploading patched applications on your device for real testing. Note that if this plugin won't connect to the phone (e.g. you have a Nokia 6600 and an unsupported BlueTooth stack) you will have to disable the Connection Manager and use InfraRed (see section 2.1).
- Please also note that if this plugin do not still work with your S60 phone, it can be a limit of the plugin which has problems with Symbian 8 and 8a phones (e.g. 6680, 668, N70, 7610, 6670). You can use instead OBEX as transfer protocol: install an OBEX server on the phone, for example *Flander FileExplorer* [46] and an OBEX client on the PC, for example the *VNavigator Siemens Obex File System* [47] Total Commander plugin (which works with Nokias too).
- *UnMakeSIS* [6] is a freeware tool (older versions at least) for unpacking a Symbian .sis file. With UnMakeSIS, analyzing and extracting Symbian .sis files is relatively easy. When using UnMakeSIS, it's important to set your screen resolution to 1024x768; the program doesn't dynamically adjust to your screen size and you may find that you can't use the functions needed to extract the .sis file⁵.
 - *uNsis by the3sky* [7], very recent tool is also able to manage SISX files⁶. This tool is essentially a graphic interface of the *NewLC SISInfo* tool, written with Python. Then it also requires installing the Python distribution. The whole procedure and download is described here: <http://www.niksula.cs.hut.fi/~jpsukane/sisinfo.html>. If the interface doesn't work for you I suggest using the python script directly, has an easy interface, like any other unpacking tool. You can anyhow transform it into an exe using a Python to Exe converter.
 - *UnSIS* by NewLC, a Perl script to extract files from a SIS archive, <http://www.newlc.com/UnSIS.html>

Tools to Create SIS files

Essentially I use only one tool, *MakeSIS* [8], a really nice tool to create new SIS files. It will be useful at the end of the patching process to create a new SIS file to distribute.

2.2.2 SIS File Structure

Whatever tool you will use you will find into a SIS archive these information.

- Languages used.
- app's UID.
- Supports Series 60 version info.
- Files to install.
- Files to Run while install.
- Certificates

The language and app's UID is contained into the SIS header, which is composed of two lines as follows:

```
&EN
#{ "CoolMMS" }, (0x101FBADE), 1, 0, 6
```

For example an "&EN" says that the language is English - strictly speaking UK English. A few of the other languages are: PO Portuguese, NO Norwegian, FR French, SP Spanish, GE German, RU Russian, IT Italian, DU Dutch, SW Swedish, DA Danish, FI Finnish...

⁵ atzplzw pointed me to his latest releases of the tool (which now supports SISX format), but not more free.

⁶ Must be installed in the path he proposed otherwise doesn't work

The second line contains further information. The string is the name of the app as it is used during installation. The hex number is a UID: it is actually the UID of the .sis file itself, and is used to tell if a package is already installed. Normally if you will release a patch for an existing application you should use the same UID of the original application.

The final three numbers are respectively the major and minor version numbers of your application.

The Supports Series S60 version info has this format

```
(0x101F6F88), 0, 0, 0, {"Series60ProductID"}
```

This states that the application is for series 60 version 2

The Files to Install files are stored using a special format. The "!" in the name indicates that the user can choose which drive to use. I can place on drive C: (PHONE MEMORY) or E: (MEM CARD). if instead of "!" you see C or E the file will be installed to the phone memory or Memory Card directly, without options.

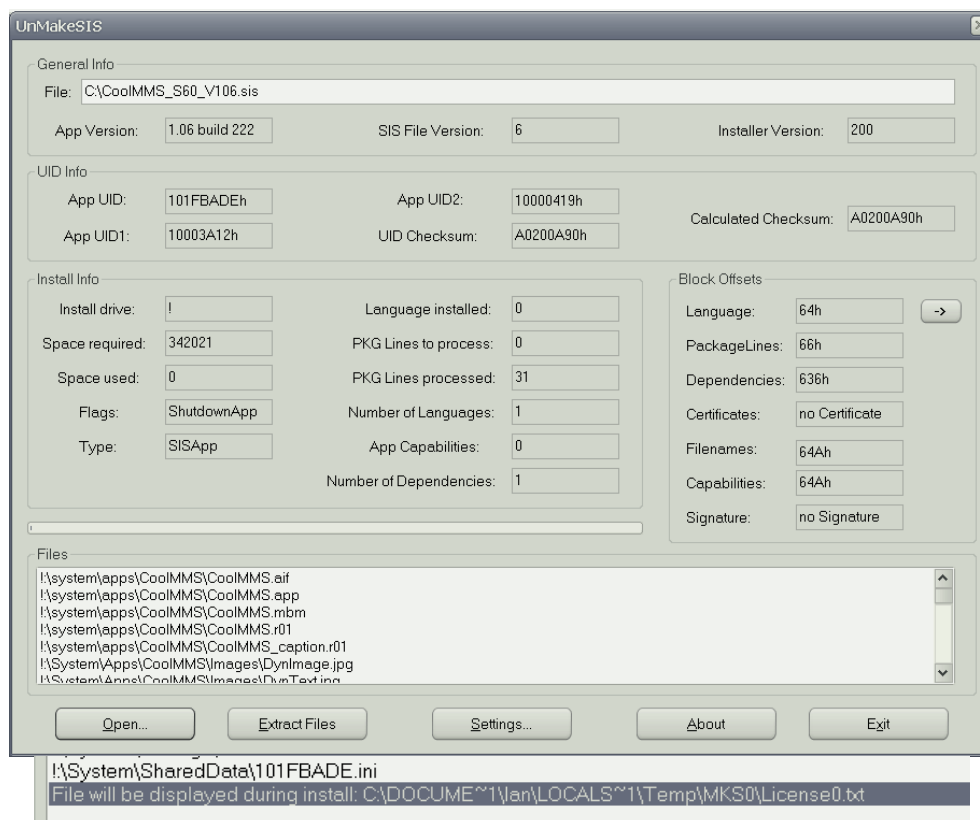
The section "Files to Run while install" contains files marked with special flags into the SIS file and clearly shown by unpacking SIS tools.

There are different types of installed files, these are the most common [12, 13, 14, 38].

- **APP GUI applications**
 - End user applications, accessible from applications menu
 - Each application must have own directory under System\apps in some drive in order to be visible for user in application launch menu
- **MDL Recognizer components**
 - Provide file association services for rest of the OS
 - Also the most common method for starting applications at boot
 - Start automatically at boot or from inserted memory card
 - Must be located on System\recogs directory
- **MBM SymbianOS Multi BitMap Image**, a file containing several bitmaps
- **AIF Application Information Files.**
 - Program needs two things, a nice icon and a nice name. Both these goals can be achieved by the AIF (Application Information File).
- **RSC, RSS, RSP, Rxx Application resource files.**
 - Many elements of a user interface, for instance dialogs, the button bar and menu panes, generally known as resources, are defined using a Symbian OS-specific resource language rather than directly in C++. Resources are expressed in text files which are then compiled into compressed binary application resource files as part of the build process. An advantage of separating the resources from the C++ code is that the resource files can be edited separately, for instance by a localization team, without any impact on the rest of the program. Each application has a resource file (with the same name as the application itself), which is also used by the application framework.
The resources might also be directly linked to the program; this is absolutely the simplest situation because IDA can directly resolve strings and resources call.
 - RSS are the source of resource files
 - RSC are the compiled version of the same files
 - RSP files are multilingual resource files
 - Rxx files, where xx is a two digits number (like .r01, r02 etc) are resources files too

Finally Certificates should be used to certify that an application has been modified, but nowadays I never seen a single certified application, even the originals and those distributed by big companies.

For example this is the result for one of the application used later on (CoolMMS)..



2.2.3 A Note on SISX Files

The SISX files are used by the S60 3rd Edition and contain a completely different architecture. If you manage to unpack them (for example using SISInfo by NewLC) you will find into the archive .exe and .dll files. These are obviously not Win32 exe or PE format files. The format is another specific for Symbian. Unfortunately here is where IDA fails to work, currently this format is not supported (with IDA 5.0, but IDA 5.1 supports Symbian 9.0).

Here are some links about the specifications:

- New Symbian OS 9 Executable File Format (E32Image), <http://www.antonypranata.com/articles/e32fileformatv9.html>
- Symbian OS Executable File Format (E32Image), <http://www.antonypranata.com/articles/e32fileformat.html>
- Reading Import Section of Symbian OS Executable File (E32Image), <http://www.antonypranata.com/articles/e32importsection.html>

I found on the market some free interesting unpackers for SISX files, of course more will appear:

- **SISXplorer** from Symbian-Toys [39], which is moreover totally free (written using .NET).
- **SISWare**, a very complete free program that also handle Symbian 9.x SIS files [40]

2.3. Other Useful Tools

There are other tools which are useful to understand how the program works.

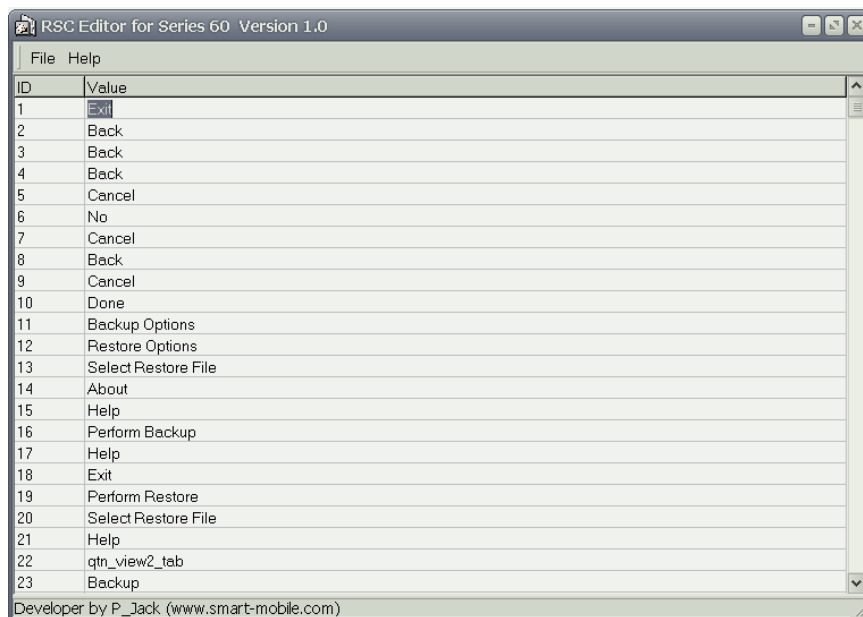
- **Resource Decompiler**. The only two efficient tools I ever seen are +Phantasm ERL [10] and RSC Editor by P_Jack [11] which can decompile the *.rsc files (ERL anyway incorrectly manage control characters).

The output you would get with ERL is a text file with a list of strings and references like the following (taken from one of the applications later examined, SpriteBackup)

```
ResNr  ResHex  ResourceID  ResType      Resource      (Note: ResType is only a suggestion!)
1      1h      8F6F001    Unknown      ''
```

2	2h	8F6F002	String	Sprite Backup
3	3h	8F6F003	HotKeys	
4	4h	8F6F004	HotKeys	⌘e
5	5h	8F6F005	Defaults	"/"/"/
6	6h	8F6F006	Buttons	P^Options⌘Exit
7	7h	8F6F007	Buttons	P^Options;⌘Back
8	8h	8F6F008	Buttons	PStart;⌘Back
9	9h	8F6F009	Buttons	PStart;⌘Back
10	Ah	8F6F00A	Buttons	P^Select⌘Cancel
11	Bh	8F6F00B	Buttons	P^Yes⌘No
12	Ch	8F6F00C	Buttons	P^OK⌘Cancel
13	Dh	8F6F00D	Buttons	P^OK⌘Back

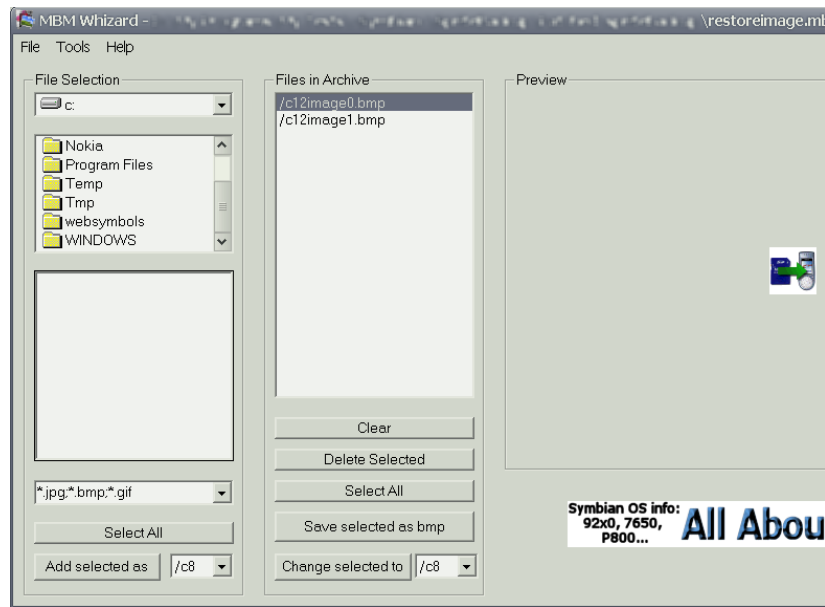
With RSC Edit instead you will get this (note that what RSC Edit calls ID is called ResNr of ERL ;-)):



This information can be used to see with IDA where the application uses a resource.

Unfortunately these tools seem to be able to reverse only the resources format of OS5-6 and not the new one for OS7. Despite deep searches I were not able to find any updated program!

- **MBM wizard** [15], this program is able to decompress and convert back to normal formats and also update MBM files (useful if you want to modify some graphic of the final patched application). At [40] there's also another interesting MBM editor you can try.



2.4. Useful Tools working on the phone

There are some programs which might be interesting to have installed on the phone, to dig a little more the application, the file system and keep safe the phone.

- Backup Program. I suggest using any backup program you like before starting playing with applications, a freeze of the phone might happen. I suggest using SpriteBackup (see § 3).
- FExplorer, a great file explorer, completely free.



- About, a great source of information of your phone, what OS, memory and status.



- AppMan, another excellent process explorer, to explore processes, thread, open files,



- WibuCapture an excellent program to capture screens from the phone, the one I used for this tutorial.



- On device disassemblers. Not much is available, the only one I found is DisAsm, an unfinished program available at [44], and it doesn't work on S60 phones.

2.5. Analysis done by IDA

IDA as said recognizes the Symbian/EPOC file format (not the Version 9.x) and then is able to do a static dead listing of the program. It should now be clear that the thing you should give to IDA is the *.app application file. If the program directly contains the resources, lot stored into an external resources file, the analysis will be complete and IDA will be able to resolve strings and resources usage. These cases are simpler than any other application (and we will start from an example like this).

If the resources are external you will have to coordinate the work with one of the resource decompiler introduced before.

2.5.1 For those completely unaware of what IDA is

This document, for clear reasons cannot cover the most essential things on IDA. I anyway suggest for those of you completely new to the IDA way of working to read these nice tutorials I also mirrored on our site:

- <http://arteam.accessroot.com/tools/symbian/IdaTut.zip>
A public (despite the header) document on generic way of working with IDA. Consider that it is not updated to the latest IDA release so some screenshots might be different. Anyway the basic organization of the IDA interface is the same since the early versions.
- <http://arteam.accessroot.com/tools/symbian/Quick Start Guide IDA 3.38.zip>
A quick start on an old version of IDA, but the basic concepts as I already said are still good.
- <http://arteam.accessroot.com/tools/symbian/IDA Pro Shortcuts.rar>
A handful list of shortcuts with IDA

I think that the three documents above should be enough to start understanding what IDA is and what it can do.

2.5.2 Improving the IDA Analysis adding missing IDS

Unfortunately IDA, doesn't have all the SymbianOS system libraries signatures. Then the analysis done is not complete because it lacks of several system calls to APIs. For example this is what we get with the CrackMe I release contemporarily with this tutorial [30]. It is using the AVKON libraries, which is the most important library offering APIs for creation of dialogbox, listbox, messagbox and so on.. I will explain more about this later on. Now concentrate on the different analysis results we have.

Figure 4 and Figure 5 clearly show the difference between the two analyses: the first doesn't resolve the AVKON and AKNOTIFY imports while the second one is able to resolve them.

Note: functions within a ".dll" are referenced by ordinal number, the Symbian logic expects the compiler to provide a translation function (a stub) that maps a given function name into the corresponding ordinal number.

Names reported in Figure 5 are undecorated names, resolved by IDA automatically in the code. Figure 6 and Figure 7 shows the result of this undecoration. It becomes clear that the AVKON_12 stub is a call to CAknBatteryPane::CAknBatteryPane the constructor of the class CAknBatteryPane. You can use the online reference library to understand what a specific API does, using for example [18], but consider that as usual some API are not documented just because developers shouldn't ever use them⁷.

Address	Ordinal	Name	Library
10006D40	4	AKNNOTIFY_4	AKNNOTIFY
10006D44	5	AKNNOTIFY_5	AKNNOTIFY
10006D48	9	AKNNOTIFY_9	AKNNOTIFY
10006D4C	10	AKNNOTIFY_10	AKNNOTIFY
10006D50	28	AKNNOTIFY_28	AKNNOTIFY
10006D54	30	AKNNOTIFY_30	AKNNOTIFY
10006D58	36	AKNNOTIFY_36	AKNNOTIFY
10006D5C	52	AKNNOTIFY_52	AKNNOTIFY
10006D60	53	AKNNOTIFY_53	AKNNOTIFY
10006D64	61	AKNNOTIFY_61	AKNNOTIFY
10006D68	3	AppFullName__C15CAppApplication	APPARC
10006D6C	6	Capability__C12CAppDocument	APPARC
10006D70	27	GlassPictureL__12CAppDocument	APPARC
10006D74	71	ValidatePasswordL__C12CAppDocument	APPARC
10006D78	12	AVKON_12	AVKON
10006D7C	30	AVKON_30	AVKON
10006D80	32	AVKON_32	AVKON
10006D84	37	AVKON_37	AVKON
10006D88	42	AVKON_42	AVKON
10006D8C	60	AVKON_60	AVKON

Figure 4 –Analysis results of resolved Import names without any additional Symbian IDS file

Address	Ordinal	Name	Library
10006D40	4	imp__imp_??1CAknGlobalMsgQuery@@@UAE@XZ	AKNNOTIFY
10006D44	5	imp__imp_??1CAknGlobalNote@@@UAE@XZ	AKNNOTIFY
10006D48	9	imp__imp_??1CAknPopupNotify@@@UAE@XZ	AKNNOTIFY
10006D4C	10	imp__imp_??1CAknSignalNotify@@@EAE@XZ	AKNNOTIFY
10006D50	28	imp__imp_?CancelMsgQuery@CAknGlobalMsgQuery@@@QAEXXZ	AKNNOTIFY
10006D54	30	imp__imp_?CancelProgressDialog@CAknGlobalProgressDialog@@@QAEXXZ	AKNNOTIFY
10006D58	36	imp__imp_?NewLC@CAknGlobalConfirmationQuery@@@SAPAV1@XZ	AKNNOTIFY
10006D5C	52	imp__imp_?NewLC@CAknGlobalProgressDialog@@@SAPAV1@XZ	AKNNOTIFY
10006D60	53	imp__imp_?NewLC@CAknIncallBubble@@@SAPAV1@XZ	AKNNOTIFY
10006D64	61	imp__imp_?ProcessFinished@CAknGlobalProgressDialog@@@QAEXXZ	AKNNOTIFY
10006D68	3	imp__imp_AppFullName__C15CAppApplication	APPARC
10006D6C	6	imp__imp_Capability__C12CAppDocument	APPARC
10006D70	27	imp__imp_GlassPictureL__12CAppDocument	APPARC
10006D74	71	imp__imp_ValidatePasswordL__C12CAppDocument	APPARC
10006D78	12	imp__imp_??0CAknBatteryPane@@@QAE@XZ	AVKON
10006D7C	30	imp__imp_??0CAknDouble2GraphicStyleListBox@@@QAE@XZ	AVKON
10006D80	32	imp__imp_??0CAknDoubleGraphicStyleListBox@@@QAE@XZ	AVKON
10006D84	37	imp__imp_??0CAknDurationQueryDialog@@@QAE@AAVTIntervalSeconds@@@ABW4TTone@CAknQueryDialog@@@Z	AVKON
10006D88	42	imp__imp_??0CAknEdwinSettingPage@@@IAE@PBVTDesC16@@@HHHH@Z	AVKON
10006D8C	60	imp__imp_??0CAknIconArray@@@QAE@H@Z	AVKON

Figure 5 - Analysis results of resolved Import names with additional Symbian IDS file

⁷ Not causally it's the case of CAknBatteryPane, which is not documented on official SDK documentation, anyway it is documented in external internet sources; you can try with other APIs in your program and see what's the result..

```
.text:10005BCC ; :::::::::::::: S U B R O U T I N E ::::::::::::::
.text:10005BCC
.text:10005BCC ; Attributes: thunk
.text:10005BCC AUKON_12 ; CODE XREF: sub_10000104+60↑p
.text:10005BCC          LDR    R12, =__imp_AUKON_12
.text:10005BD0          LDR    R12, [R12]
.text:10005BD4          BX     R12
.text:10005BD4 ; End of function AUKON_12
.text:10005BD4 ; -----
.text:10005BD4 ;
```

Figure 6 – Analysis of one function without new IDS

```
.text:10005BCC ; :::::::::::::: S U B R O U T I N E ::::::::::::::
.text:10005BCC
.text:10005BCC ; Attributes: thunk
.text:10005BCC ; __declspec(dllimport) public: __thiscall CAknBatteryPane::CAknBatteryPane(void)
.text:10005BCC __imp__0CAknBatteryPane__QAE_XZ ; CODE XREF: sub_10000104+60↑p
.text:10005BCC          LDR    R12, =imp___imp___0CAknBatteryPane__QAE_XZ
.text:10005BD0          LDR    R12, [R12]
.text:10005BD4          BX     R12
.text:10005BD4 ; End of function CAknBatteryPane::CAknBatteryPane(void)
.text:10005BD4 ; -----
.text:10005BD4 ;
```

Figure 7 - Analysis of one function with new IDS

But how to create those IDS files? I had to create on my own using the `idsutils` for IDA and the following procedure (with a little of automation).

1. Download the Symbian SDK from the Nokia site (for example from [3])
2. Download ActivePerl and install it, as requested to install the SDK (see the readme inside)
3. Install the whole SDK
4. Go to the `symbian\S60_3rd_FP1\Epoc32\release\winscw\udeb\` and copy all the *.lib files (A mirror is also available at http://arteam.accessroot.com/tools/symbian/Symbian_7.0_S60_LIB_files.rar)
5. Launch on any of it the lib files the program `ar2idt.exe` or use an automation tool like for example `lst2mlt` for Total Commander.

```
Command: %COMMANDER_PATH%\Tools\lst2mlt.exe /X:"%COMMANDER_PATH%\Tools\ar2idt.exe"
Parameter: /L:"%L"
```

Note that the use of `lst2mlt` is optional: I just used it to automate the launching of `ar2idt` over several files, normally you will not repeat this operation and just use the IDS files I provided.

6. Once all the `ar2idt` instances finished launch on all the created idt files the `zipids.exe` program in order to create the IDT files. You can use another time an automation tool for Total Commander, as before

```
Command: %COMMANDER_PATH%\Tools\lst2mlt.exe /X:"%COMMANDER_PATH%\Tools\zipids.exe"
Parameter: /L:"%L"
```

7. Once all `zipids` instances finished, copy all the IDS files into the `\IDA Pro Disassembler\ids\epoc6\arm\` folder. Take care to not overwrite the existing files, because usually are made better, even if some exports are missing..
8. Now the next time you do a new analysis with IDA, it automatically applies the new signatures.

If you want you can compare already existing ids files using the `zipids -u <file.ids>` command. Compare for example the IDT files given with IDA and those you generated with above steps.

IDS files are text files that can be edited to adjust functions but takes long to fix all, so better leaving as the tool generated them. Anyway in the official `idsutils` distribution you can find how the ids file is made and then start from that editing.

If you want to get an already prepared distribution of these files get them at link [17].

2.5.2.1 The EFD utility

The author of IDA (Guilfanov) released some time ago, through his blog an interesting utility, EFD [41]. This utility, despite not underlined into the Guilfanov' post on the blog, supports also Symbian libraries (see Figure 8) and you can use it in place of ar2idt⁸:

For example you can use this command-line to get the avkon.idt file

```
efd -t avkon.lib
```

```
C:\WINDOWS\system32\cmd.exe
C:>efd
Extensive File Dump. Version 2.62 Copyright (c) 1995-2005 by Ilfak Guilfanov
Supported formats: EXE, NE, LE, LX, PE, NLM, XCOFF, COFF, OMF, DBG, PRC, PEF,
OS9, N64, PSX, EPOC, AR, AMIGA, ELF, ECOFF, HP SOM, GEOS, OLE2, AIF, AOF,
AOUT, PE+, OMF166, Mach0, XE/XBE, JPG, CIFF, TMOBJ, MRW, TIFF, MPG, CWLIB,
XCP.DAT
This program is FREEWARE. Please send all comments to <ig@hexblog.com>
Usage: efd [-sw] exec-file
        -C create CIL declaration file (only .Net files)
        -c interpret as COFF file (-c2 - big endian)
        -f don't show fixup information
        -i interpret as IBM OMF file
        -l show line number information
        -m show MS DOS header
        -o interpret as OMF file
        -p interpret as PE file
        -P interpret as PRC file
        -r don't show resource table
        -s don't show symbol table
        -t create IDT files from import library
        -u### undecorate name in AR lib and create the specified output lib
        -x extract modules from libraries (-x2-use file names)
        -exonly export information only (PE files)
```

Figure 8 - efd command line help

Note: an important note on the ids created for this tutorial.

Nokia is used to add and remove some imports name and assign previous ordinals to new methods on different releases of the SDK. So it might happen (actually it happens) that you find an ordinal assigned to method A into a version of the SDK libraries and then the same ordinal assigned to method B into the following version.

This makes IDS less useful because IDA only read and applies the information into the IDS files, without any further reasoning. So if you apply wrong IDS files to a program compiled with another version of the library you will see wrong imports reported by IDA. To use ids is then important to use the correct ids for the application sdk used for the application.

2.5.3 Improving Imports readability: undIDC tool

The next problem we have to solve is the fact the imports coming from the IDS files I just created are decorated (not in the usual C/C++ prototypes format) and thus almost unreadable and, that matters most, sometime very long, depending on the number of arguments (technically you should speak of demangling import names).

For example, always using the same example above we have that the import #37 of library AVKON has the following decorated name

```
10006D84 37
??0CAknDurationQueryDialog@@QAE@AAVTTimeIntervalSeconds@@ABW4TTone@CAknQueryDialog@@@Z
```

⁸ Actually it supports a lot of library formats, as you can see from Figure 8.

Looking also at Figure 5, it is clear that this is not the better solution, to let the whole code be readable.

```

idata:10006D84 : __declspec(dllimport) public: __thiscall CAknDurationQueryDialog::CAknDurationQueryDialog(class YTimeIntervalSeconds &, enum CAknQueryDialog
idata:10006D84 : IMPORT      Import CAknDurationQueryDialog::CAknDurationQueryDialog::CAknDurationQueryDialog

.text:10006710      DCD   _Draw_CFormGraphicListBoxData__UBEXUTListItemProperties__AAUCWindowGc__PBUTDesC16__ABUTRect__HABUTColors_CFormattedCellListBo
.text:10006714      DCD   _CAknDurationQueryDialog__QAE_AAUTTimeIntervalSeconds__ABW4TTone_CAknQueryDialog__Z : CAknDurationQueryDialog::CAknDuratio
.text:10006718      DCD   _HandleMarkableListDynInitMenuPane_AknSelectionService__SAXHPAUCEikMenuPane__PAUCEikListBox__Z : AknSelectionService::Handle
.text:1000671C      DCD   _HandleListBoxEventL_CAknSelectionListDialog__MAEXPAUCEikListBox__W4TListBoxEvent_MEikListBoxObserver__Z : CAknSelectionList

```

IDA undecorates the names, but leaves the decorated names when the code is calling the relative import, like for example this snippet of code (taken from the ARTeam CrackMe for Symbian 1.0 [30])

```

sub_100063AC
SUB     R0, R0, #0x14
B       HandleSideBarMenuL_9CEikAppUi::iRC6TPointiPC15CEikHotKeyTable : CEikAppUi::HandleSideBarMenuL(int,TPoint const &,int,CEikHotKeyTable const *)
; End of function sub_100063AC

```

This is not fair. You have to do two things:

1. modify the demangling IDA options like in Figure 9:

```

sub_100063AC
SUB     R0, R0, #0x14
B       CEikAppUi::HandleSideBarMenuL(int,TPoint const &,int,CEikHotKeyTable const *)
; End of function sub_100063AC

```

2. modify the imports raw names so as to met have a faster browsing of the IAT of Symbian applications (which is really important in order to view faster which Symbian API the program is calling). To accomplish this task I had to write a little tool to undecorate raw Import names, resolving the problem using custom conversion syntax.

2.5.3.1 Using the undIDC custom tool to simplify IDA Imports demangling

Note: The program is available at [35] and the sources at [36] (extremely badly written I know!!). I will only explain here how to use the tool and what it does. I have done several tests, but the tool is far from perfect and mature, the source code is not too optimized. So use the sources also if you want to improve the tool, or if you want to do a new one using the same algorithm/idea.

1. First of all be sure to have these undecoration/demangled settings with IDA (see Figure 9 and Figure 10).

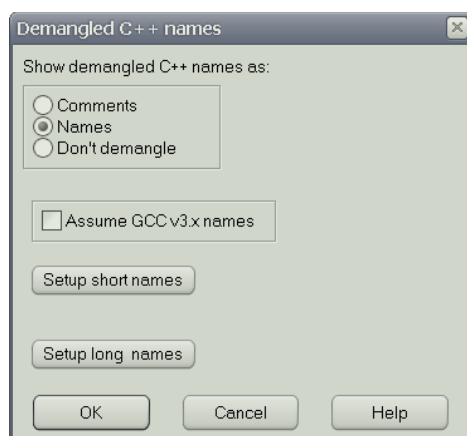


Figure 9 – IDA Options -> Demangled Names menu

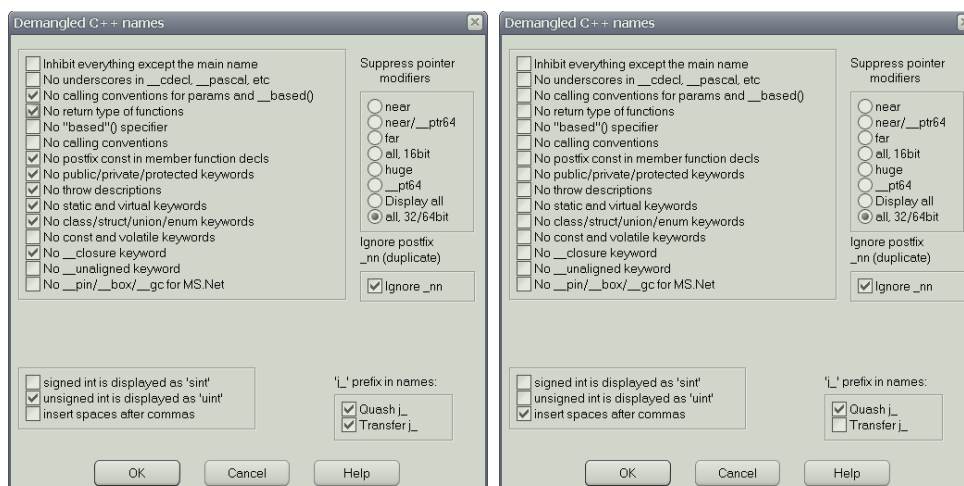


Figure 10 – Respectively “Setup short names” and “Setup long names” views

- Second, select the Dump “Database to IDC file..” function of IDA, like in Figure 11.

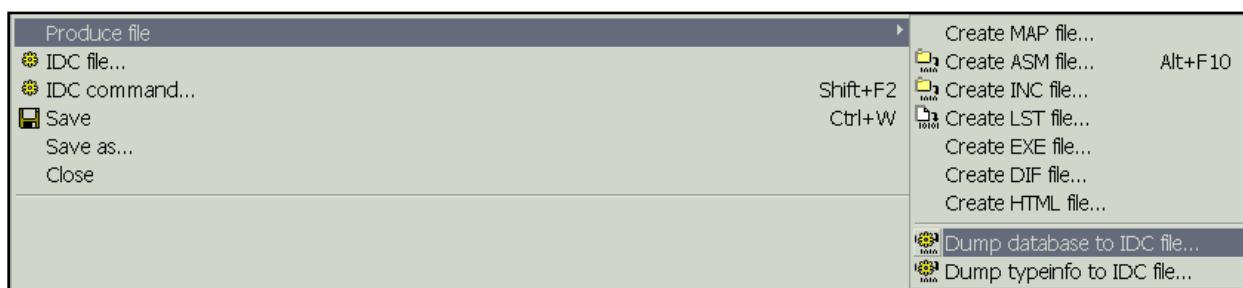


Figure 11 - “Dump database to IDC file” IDA menu

Save the file with the same name of the application you are studying. E.g. AknExQuery.idc

- Open the undIDC program I did (it's a dos program) [35] using the following command-line:

```
undIDC AknExQuery.idc
```

the program creates a new IDC file with a suffix “UND” in the name, so following previous e.g. it produces AknExQueryUND.idc

- Execute the new script into IDA. The new script must then be executed into IDA and the new names will be simplified.

When terminated IDA reports a custom message in the status window:

```
mIDA: Can't be loaded (not a PE file or debugger mode)[!] De
UndecorateSymbol: Done renaming the imports and stubs!
AU: idle | Downl Disk: 6GB |
```

- Close and reopen the *.idb file and you now get the imports renamed as wanted.

10006D80	32	CAknDoubleGraphicStyleListBox_CAkndoubleGraphicStyleListBox_C164D23A5F837F2EA54351676E6944DF	AVKON
10006D84	37	CAknDurationQueryDialog_CAkndurationQueryDialog_63540E6BBA366C7C5512CD76D4A31812	AVKON
10006D88	42	CAknEdwinSettingPage_CAknedwinSettingPage_1DC2A4C7ABCB36E0483B0CCC75E0B956	AVKON
10006D8C	60	CAknIconArray_CAkniconArray_8ACE454DEFC90857E15483031B317710	AVKON

There's a new name for function at 10006D84, which explicitly contains the class and method without badly human-reading decorations.

The import table was something like the following:


```

:idata:10006D80      IMPORT _Cba_CaknAppUi__QAEPRUCEikButtonGroupContainer__XZ
:idata:10006D80      ; DATA XREF: .text:off_10005FF8to
:idata:10006D84      IMPORT __imp__ComparePasswords_CaknAlphaPasswordSettingPage__MBEHABUTDesC16__0M4TknPasswordMatchingMode_CaknPasswordSettingPage
:idata:10006D84      ; DATA XREF: .text:off_10005E18to
:idata:10006D88      IMPORT _ConstructFromResourceL_CaknIntegerEdwin__UAEAXAUTResourceReader____Z
:idata:10006D88      ; DATA XREF: .text:off_10005FD8to
:idata:10006D8C      IMPORT __imp__ConstructL_CaknIntegerSettingPage__UAEXXZ
:idata:10006D8C      ; DATA XREF: .text:off_10005C48to
:idata:10006DC0      IMPORT __imp__ConstructL_CaknNavigationControlContainer__QAEXXZ
:idata:10006DC0      ; DATA XREF: .text:off_10005C18to
:idata:10006DC4      IMPORT _ConstructL_CaknNavigationDecorator__QAEXXZ
:idata:10006DC4      ; DATA XREF: .text:off_10006138to
:idata:10006DC8      IMPORT _ConstructL_CaknNumericPasswordSettingPage__UAEXXZ
:idata:10006DC8      ; DATA XREF: .text:off_10005CD8to
:idata:10006DCC      IMPORT _ControlFactory_CaknLibrary__SAP6A__AUSeikControlInfo__H_ZXZ
:idata:10006DCC      ; DATA XREF: .text:off_10005FC8to
:idata:10006DD0      IMPORT _ConvertToUisualAndClip_AknBidiTextUtils__SAPAUHBufC16__ABUTDesC16__ABUCFont__HWW4TParagraphDirectionality_1_UTChar____Z
:idata:10006DD0      ; DATA XREF: .text:off_10005C78to
:idata:10006DD4      IMPORT _ConvertToUisualAndWrapToArrayL_AknBidiTextUtils__SAPAUHBufC16__ABUTDesC16__HABUCFont__AAU__ArrayFix_UTPtrC16____W4TPara
:idata:10006DD4      ; DATA XREF: .text:off_100060E8to
:idata:10006DD8      IMPORT _ConvertToUisualAndWrapToArrayL_AknBidiTextUtils__SAXAUTDesC16__ABU__ArrayFix_H__ABUCFont__AAU__ArrayFix_UTPtrC16____HW
:idata:10006DD8      ; DATA XREF: .text:off_10005C98to
:idata:10006DDC      IMPORT __imp__ConvertToUisualAndWrapToArrayWholeTextL_AknBidiTextUtils__SAPAUHBufC16__ABUTDesC16__ABU__ArrayFix_H__ABUCFont__AA
:idata:10006DDC      ; DATA XREF: .text:off_10005C28to
:idata:10006DE0      IMPORT _CopyFormattedNumber_CaknPhoneNumberGrouping__QBEAXAUTDesC16____Z
:idata:10006DE0      ; DATA XREF: .text:off_10005CA8to
:idata:10006DE4      IMPORT __imp__CountComponentControls__AknListBoxLinesTemplate_UCEikFormattedCellListBox____UBEHXZ
:idata:10006DE4      ; DATA XREF: .text:off_10005BF8to
:idata:10006DE8      IMPORT __imp__CountComponentControls_CaknBatteryPane__MBEHXZ

```

And now becomes instead:

```

:idata:10006D80      IMPORT CaknAppUi_Cba_8EC963F59DE0C5B0CFA8C931AB82566B
:idata:10006D80      ; DATA XREF: .text:off_10005FF8to
:idata:10006D84      IMPORT CaknAlphaPasswordSettingPage_ComparePasswords_044CFC54E3BA773CDFFC5325DFC9E64E
:idata:10006D84      ; DATA XREF: .text:off_10005E18to
:idata:10006D88      IMPORT CaknIntegerEdwin_ConstructFromResourceL_C05DF25F86E5AA8997EA411717F064EF
:idata:10006D88      ; DATA XREF: .text:off_10005FD8to
:idata:10006D8C      IMPORT CaknIntegerSettingPage_ConstructL_0F0FEEEDC07168580F06517A9A193089
:idata:10006D8C      ; DATA XREF: .text:off_10005C48to
:idata:10006DC0      IMPORT CaknNavigationControlContainer_ConstructL_9ACBA4396757BEF832279550C3A2B258
:idata:10006DC0      ; DATA XREF: .text:off_10005C18to
:idata:10006DC4      IMPORT CaknNavigationDecorator_ConstructL_47A4894868F1685C2EDA22F37CC90639
:idata:10006DC4      ; DATA XREF: .text:off_10006138to
:idata:10006DC8      IMPORT CaknNumericPasswordSettingPage_ConstructL_0F0FEEEDC07168580F06517A9A193089
:idata:10006DC8      ; DATA XREF: .text:off_10005CD8to
:idata:10006DCC      IMPORT CaknLibrary_ControlFactory_D3A1E6E26ED75E4AF0B695DA2D564102
:idata:10006DCC      ; DATA XREF: .text:off_10005FC8to
:idata:10006DD0      IMPORT AknBidiTextUtils_ConvertToUisualAndClip_FTAC6D18F3D31FB800909CE35FBA9F30
:idata:10006DD0      ; DATA XREF: .text:off_10005C78to
:idata:10006DD4      IMPORT AknBidiTextUtils_ConvertToUisualAndWrapToArrayL_32CB37D6C9C4FDBCE2980470853E8517
:idata:10006DD4      ; DATA XREF: .text:off_100060E8to
:idata:10006DD8      IMPORT AknBidiTextUtils_ConvertToUisualAndWrapToArrayL_A9F87053B81A90F57A6A42B218B46E87
:idata:10006DD8      ; DATA XREF: .text:off_10005C98to
:idata:10006DDC      IMPORT AknBidiTextUtils_ConvertToUisualAndWrapToArrayWholeTextL_4A57CEAE6FF9CA2903FFCD0AE1396A82
:idata:10006DDC      ; DATA XREF: .text:off_10005C28to
:idata:10006DE0      IMPORT CaknPhoneNumberGrouping_CopyFormattedNumber_8EBD76B17F9E314B746263D0DE9E0A7E
:idata:10006DE0      ; DATA XREF: .text:off_10005CA8to
:idata:10006DE4      IMPORT CEikFormattedCellListBox_CountComponentControls_AA47CFE1CD7DC5C48DFE589DE37B1B9C
:idata:10006DE4      ; DATA XREF: .text:off_10005BF8to
:idata:10006DE8      IMPORT CaknBatteryPane_CountComponentControls_B041A9D6C28EF65D6A440B8E34916A38

```

2.5.3.2 Alternative Syntax of undIDC

I implemented an alternative syntax for undIDC which uses the export of the Imports tab from IDA, to create the same result already described.

You might use this as an alternative. In some cases one or the other is faster.

Address	Ordinal	Name	Library
10006D40	4	__imp_??1CAknGlobalMsgQuery@@@UAE@XZ	AKNNOTIFY
10006D44	5	__imp_??1CAknGlobalMsgQuery@@@UAE@XZ	AKNNOTIFY
10006D48	9	__imp_??1CAknGlobalMsgQuery@@@UAE@XZ	AKNNOTIFY
10006D4C	10	__imp_??1CAknGlobalMsgQuery@@@UAE@XZ	AKNNOTIFY
10006D50	28	__imp_??1CAknGlobalMsgQuery@@@UAE@XZ	AKNNOTIFY
10006D54	30	__imp_??1CAknGlobalMsgQuery@@@UAE@XZ	AKNNOTIFY

Copy the Imports table into the clipboard and paste it into a file for example called Imports.txt. Now call the undIDC tool with this command-line

```
undIDC -paste Imports.txt
```

What the tool does is to differently parse each line of the file and then produces the file ImportsUND.idc which you can use as already described above.

Consider that this last way of creating the undecorated names is sometimes less accurate..

2.5.3.3 What undIDC does

The program search into the supplied IDC file (created by IDA) all the lines with this IDC command MakeName:

```
MakeName(0X10006DB4, "__imp_?ComparePasswords@CAknAlphaPasswordSettingPage@@MBEHABVTDesC16@@@0W4TAknPasswordMatchingMode@CAknPasswordSettingPage@@@Z");
```

But it starts to do its job only after the first found line containing the string "Imports from", like in the following example:

```
ExtLinA(0X10006D40, 1, "; Imports from AKNNOTIFY[010f9a43].DLL");
```

The new script will contain a structure like:

```
#include <idc.idc>
static main(void) {
...
MakeName(0X10006DB4, "CAknAlphaPasswordSettingPage_ComparePasswords_044CFC54E3BA773CDFFC5325DFC9E64E");
..
}
```

The names are undecorated using my own following syntax.

If the C++ prototype is something like the following:

```
<return args> className::MethodName(<arguments list>);
```

It is transformed into something like:

```
className_methodName_argsNameHash
```

where argsNameHash is calculated as the MD5 has of the following string:

```
<return args>(<arguments list>)
```

So for example for the function:

```
??0CAknDurationQueryDialog@@QAE@AAVTTimeIntervalSeconds@@ABW4TTone@CAknQueryDialog@@@Z
```

the undecorated name is:

```
public: __thiscall CAknDurationQueryDialog::CAknDurationQueryDialog(class
TTTimeIntervalSeconds &,enum CAknQueryDialog::TTone const &)
```

and the simplified name becomes:

```
CAknDurationQueryDialog_CAknDurationQueryDialog_63540E6BBA366C7C5512CD76D4A31812
```

where the HEX number at the end is the MD5 hash of the following string (obtained concatenating return arguments with function arguments):

```
public: __thiscall(class TTTimeIntervalSeconds &,enum CAknQueryDialog::TTone const &)
```

The Hash was needed to avoid problems with C++ polymorphism and duplicated simplified labels.

Finally, the undecoration is done through the external Dll "UndSymbol.dll", which exposes the function UndecorateSymbol (see the project for details). This dll calls the Dbghelp dll API UndDecorateSymbolName (again see sources to understand how). The operation is done through an external dll in order to avoid conflicts I experiences with the used compiler (Visual C++ 6.0).

2.5.4 Resolving Stubs used by the Compiler

There's another important addition to add IDA, this time thanks to the SymPorts script written by djnz [16]. This is an IDC script (IDA scripting language) for resolving imports of Symbian Epoc files.

The GCC compiler, used to create most Epoc files, generates a call stub for each imported function (for example the one shown in Figure 7). This is a common trick to reduce the number of relocations which IDA handles by naming the stub according to the imported function it points too.

However, this naming mechanism is broken (or not implemented) on the ARM platform, here's the reason for the script. All it does is to walk through the import segment and attempts to locate and rename the call stubs.

The result is something like:

Before:

```
; RLibrary::Load(TDesC16 const &, TDesC16 const &)
Load__8RLibraryRC7TDesC16T1
    LDR    R12, =__imp_Load__8RLibraryRC7TDesC16T1
    LDR    R12, [R12]
    BX     R12
; End of function RLibrary::Load(TDesC16 const &,TDesC16 const &)
```

After:

```
; Attributes: thunk
; __declspec(dllimport) RLibrary::Load(TDesC16 const &, TDesC16 const &)
__imp_Load__8RLibraryRC7TDesC16T1
    LDR    R12, =imp__imp_Load__8RLibraryRC7TDesC16T1
    LDR    R12, [R12]
    BX     R12
; End of function RLibrary::Load(TDesC16 const &,TDesC16 const &)
```

A little more readable code

2.5.5 Strings analysis of Symbian programs with IDA

Symbian uses a strings format (strings in Symbian are called descriptors) which includes the length byte as a header. Then by default IDA is not able to identify them (it uses the C null terminated strings format by default).

Strings have foreword bytes reporting the length. For example, as we also see later in Section " Reversing the first application: SpriteBackup", this is a string into the SpriteBackup program:

```

                                23 00 00 00  t.T.i.m.e...#...
.text:10010480 46 00 75 00 6E 00 63 00 74 00 69 00 6F 00 6E 00  F.u.n.c.t.i.o.n.
.text:10010490 20 00 45 00 6E 00 74 00 72 00 79 00 3A 00 20 00  .E.n.t.r.y...
.text:100104A0 49 00 73 00 45 00 76 00 61 00 6C 00 75 00 61 00  I.s.E.v.a.l.u.a.
.text:100104B0 74 00 69 00 6F 00 6E 00 45 00 78 00 70 00 69 00  t.i.o.n.E.x.p.i.
.text:100104C0 72 00 65 00 64 00 00 00                                r.e.d..
```

If you use the ASCII string style **Unicode Wide Pascal (4 bytes)** IDA recognize the strings (see section §2.5.8): just immediately switch to that format after having dropped the application into IDA.

Anyway for my experience not all the strings are in the Unicode Wide Pascal format, some are just simple Unicode, so if you select one default strings format for an application you might fail finding the others and you then will need to manually define them (see later on Section on "Reversing the first application: SpriteBackup").

I suggest to always using the Strings program from Sysinternals⁹ which reports all the strings appearing into a program in all the possible formats (Unicode or not). You can later search the offset at which the strings are in the file and define the string into IDA, to see if it is referenced.

I usually use this handy batch file integrated with Total Commander:

⁹ <http://www.microsoft.com/technet/sysinternals/Miscellaneous/Strings.msp>

```
strings.bat:
%COMMANDER_PATH%\Tools\strings.exe -q %1 >%TEMP%\strings_%2.txt
%COMMANDER_PATH%\Tools\lister.exe %TEMP%\strings_%2.txt
del /F %TEMP%\strings_%2.txt
```

Launched from Total Commander with a button like this:

```
Command: %COMMANDER_PATH%\Tools\strings.bat
Parameter: %P%N %O
```

2.5.6 Using desquirr decompiler

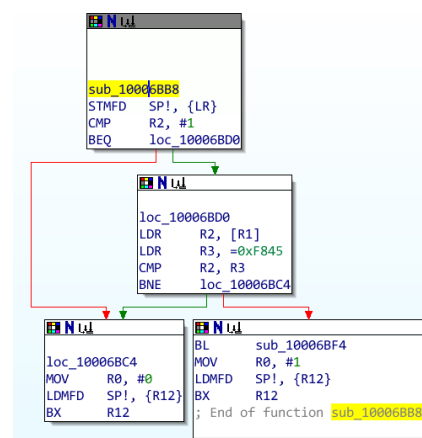


Desquirr (<http://desquirr.sourceforge.net/desquirr/>) is a decompiler wrote for IDA as part of a doctoral thesis and released for free as an IDA plugin. It "simply" decompiles the piece of code selected or the routine under the cursor. It supports both Intel and ARM assembler so it can be used for normal PC programs and for ARM based programs (Windows Mobile or Symbian). The quality of decompilation is not so excellent or so high level, as a consequence its usefulness for a CISC machine like the Intel processors is limited, but for more complex

assembler like that for ARM (ARM is a RISC CPU) it can help a lot, helping to understand for the novice the structure of the assembler instructions. Once you get used to the assembler language you won't use it anymore anyway.

The result of the decompilation goes into the log window of IDA and then moreover is not that much readable. You usually have to copy it out into a normal text editor and then easily view it.

For example the following is a function of a program we will discuss later, how it looks in IDA and a result of Desquirr: the result helps understanding what the code does and can be used for the reversing process.



```
10006bb8 sub_10006BB8:
10006bb8 /* push LR */
10006bbc Cond = R2 - 1;
10006bc0 if (Cond == 0) goto loc_10006BD0;

10006bc4 loc_10006BC4:
10006bc8 /* pop R12 */
10006bcc R12(0, R1, R2, R3);

10006bd0 loc_10006BD0:
10006bd0 R2 = * R1;
10006bd4 R3 = 0xf845;
10006bd8 Cond = R2 - R3;
10006bdc if (Cond != 0) goto loc_10006BC4;
```

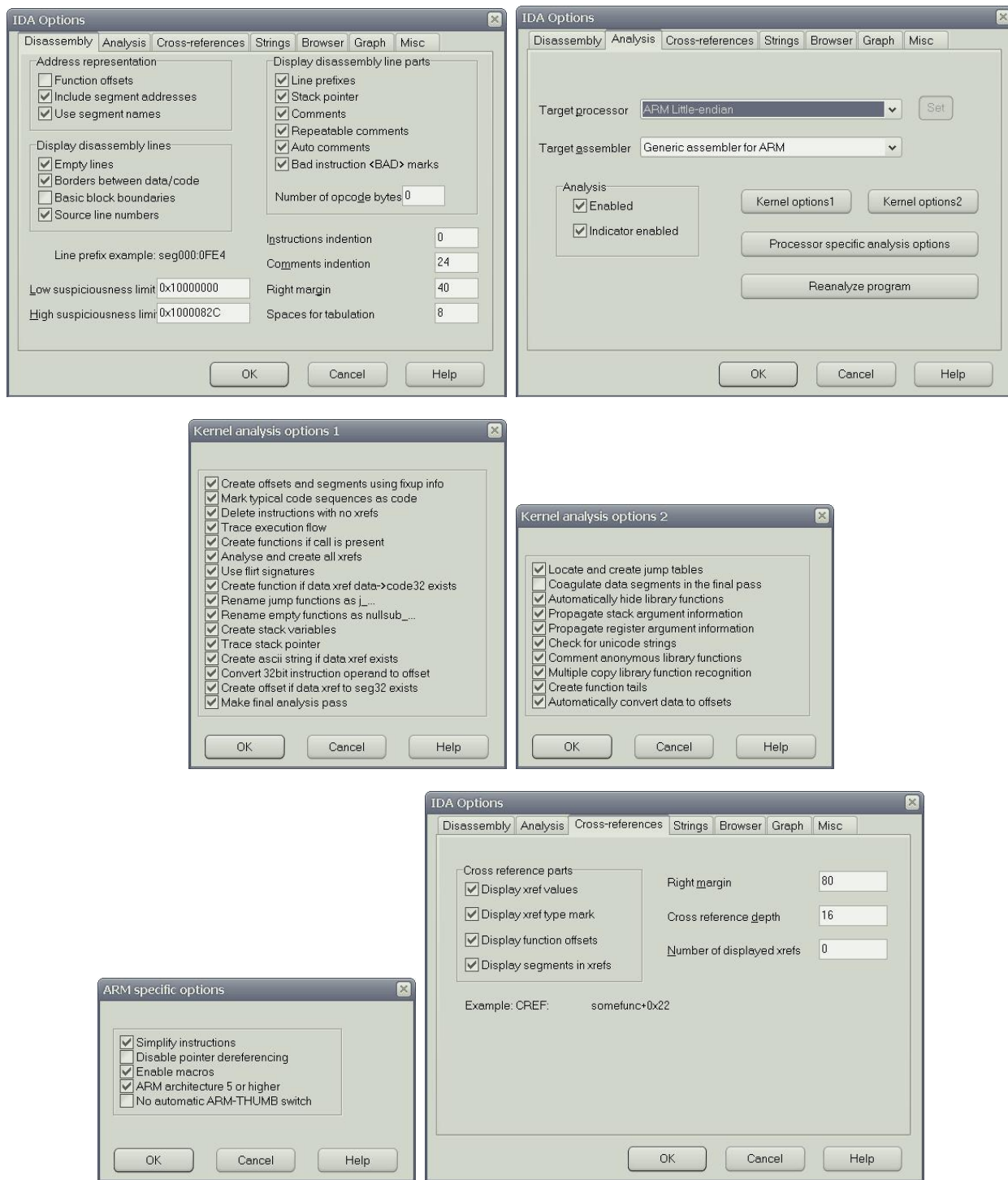
2.5.7 Reassume of the steps required to get your IDA dressed for Symbian

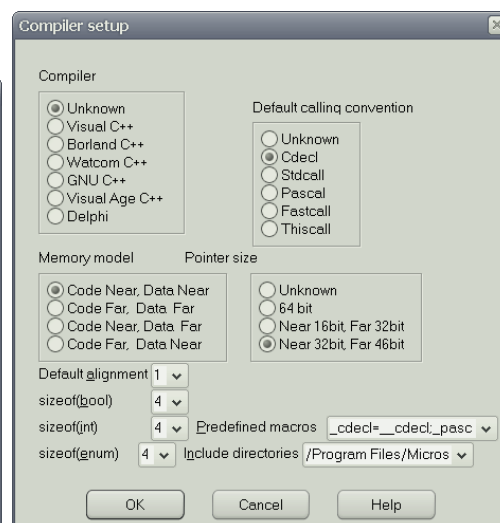
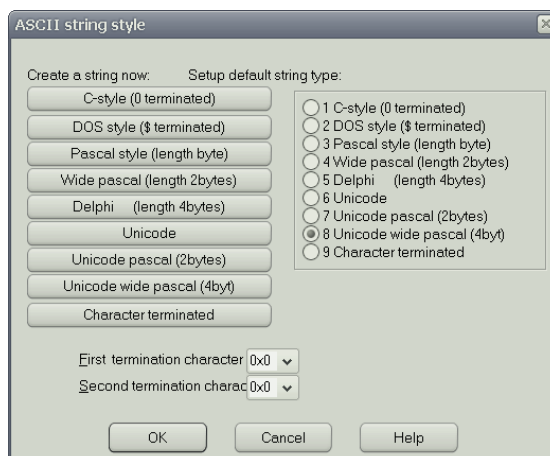
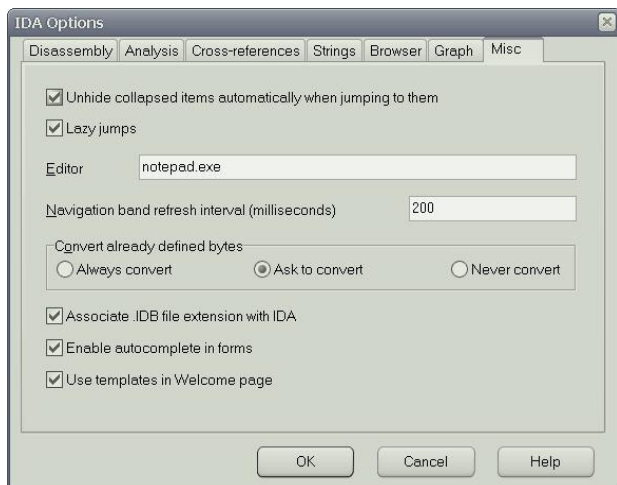
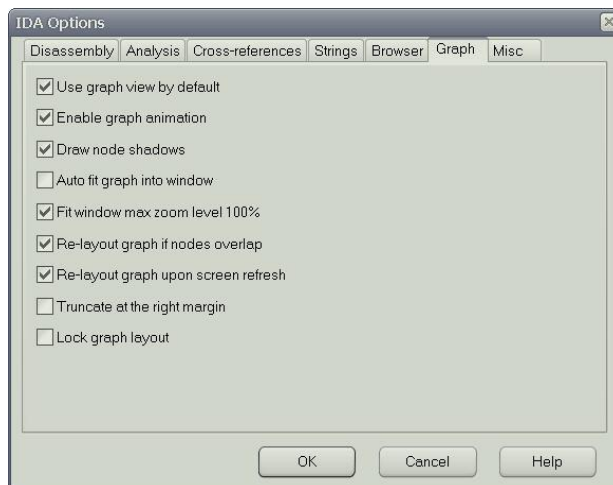
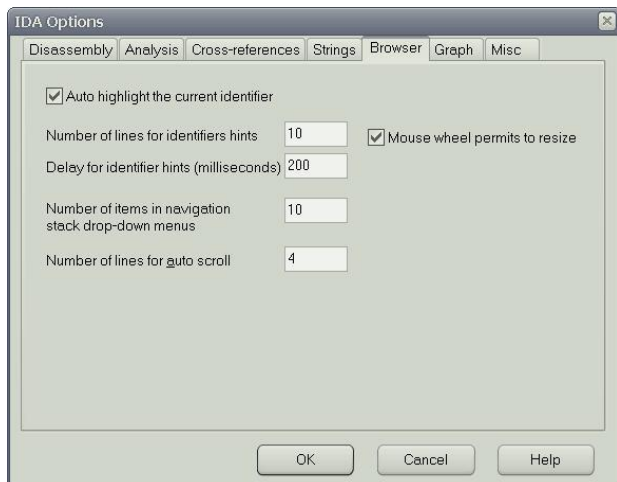
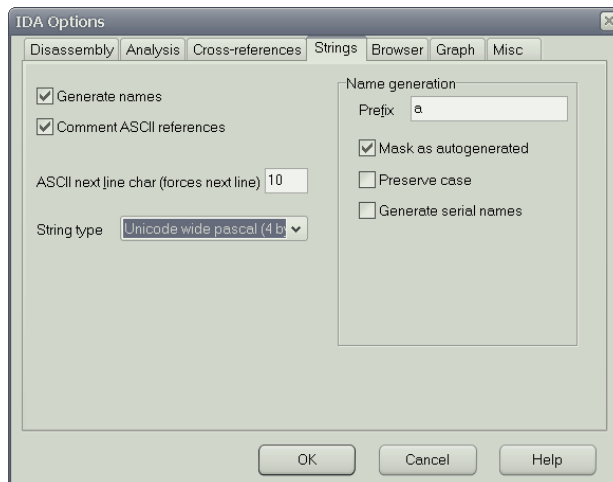
We did a lot of additions to IDA an fine tuning of settings, so you might at this stage have lost the way for what it's really needed to do and what's optional. I will reassume here the main steps.

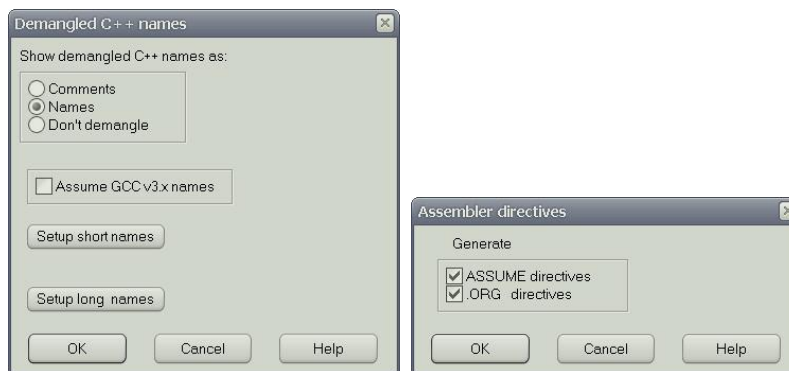
1. Install the new IDS files required to correctly identify the additional Symbian system DLLs IDA doesn't have them (Recommended)
2. Set the demangling settings as shown in Figure 9 and Figure 10 (Recommended)
3. Run the SymPorts script written by djnz [16] to correctly resolve stubs (Optional if you have recent IDAs)
4. Use the undlDC tool I wrote to improve readability of IMPORTS (Optional, just aesthetics)
5. If you are not that expert on ARM Disassembler you can also check the option "Auto comments" (see section §2.5.8), but soon you will get bored using it, because the whole code is less readable and the comments not so much interesting after all.
6. Fine tune the strings of the program defining those that are not recognized.

2.5.8 Reassume of the settings most suitable to get IDA dressed for Symbian

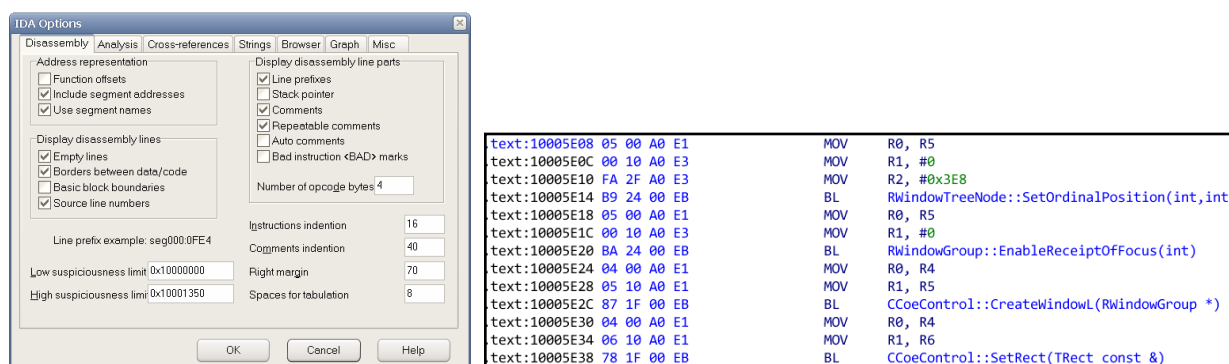
It's the moment to reassume the overall IDA settings seen so far, which are most suitable for Symbian reversing. I just report the snapshots with no further comments; they should now be self explanatory at this stage.







Note that you could also set IDA to directly show opcodes in the disassembly view so as to easily create patches, without going in the Hex view all the times:



This happens setting to 4 (the fixed length of ASM instructions) in the Disassembly option "Number of opcode bytes". It come handy when you have to find the bytes sequence in the Hex Editor to craft a patch (note: the opcodes are visible only in the Text View).

2.6. ARM Assembler

ARM processors are RISC CPUs, a Reduced Instruction Set CPU; this means (beside a lot of other things) that there are few assembler instructions and a lot of registers. This definitely means that the ASM is much more complex than the normal "x86" assembler. Anyway the assembler is the same of the WinCE/PPC tutorials you might find around, because even those platforms run on ARM processors. I (partially) wrote a tutorial at [19] reporting some basic information about this "different" assembler.

Beside this we have to learn another time the list of available opcodes to correctly patch programs.

The approach is the usual one when you have to patch programs with IDA:

1. find the opcode specification document,
2. find the proper place where to patch the application,
3. code manually the new opcode,
4. use an hex editor to write the new opcode and test again on IDA to see if it is correct.

This is the same approach described in [19] but also in [20, 21].

I will not here write any special ARM assembler course, because would take use too far, and because fast search with Google will give you hundreds of good results. Anyway in the following chapters you will find some practical examples.

Anyway you can find an excellent source of examples reading the document [22].

Once you read any tutorial I suggest printing and keeping handy the ARM Instruction Set Quick Reference Card, which reports all the instructions and syntaxes [45].

2.7. The Symbian Scene and Binary Diffing Suite

I was really wondering since the first steps into the Symbian world why there is a so urgent lack of tutorials, groups' sites and reversing forums. I had, and indeed still have, a great difficulty finding tools and tutorials. Most of the things I found were not in English or I had to use the WayBack Internet Machine to find them, because the sites were disappeared. Also usual forums such as Exetools, Woodmann, doesn't have that much.

After having scanned the network the best way I found to study was to reverse the existing cracks and searching the reasons why the guys who did those cracks did a specific modification. This has been done thanks to an excellent tool, called **eEye Binary Diffing Suite (EBDS)** [25]. I will not explain how to use this tool here, also because there are already excellent video tutorials into the distribution site, but essentially what it does is to accept two programs and give them to IDA, disassemble and then compare the resulting databases to find where the two programs differs. The results are shown graphically. The tool has been specifically thought for malware and virus analysis; with it a user can more easily find what the differences among versions of the same malware are.

Anyway what's good for analyzing malware is also good to analyze cracks and being the code analysis done by IDA, EBDS is also good to analyze any program supported by IDA. Then it was perfect for comparing original and patched Symbian applications!

2.7.1 Symbian Scene?

These are some wondering I thought about the Symbian scene, and are based on my experience.

I think that the lack of information about this world is tied to different factors:

- you have limited instruments to reverse applications (essentially only dead listings),
- the ARM assembler is quite difficult to read,
- there are several small applications, often quite frequently updated,
- the operative system often change and not all the Symbian versions are binary compatible,
- there are a lot of wannabees using the phone, even less prepared by the informatics point of view, than for the PC scene.

As a matter of facts in the PC scene tutorials, crackmes and so on are widely spread since +Fravia times and community like the one of our forum reflects the will to share knowledge. This seems not to happen for Symbian.

On the other hand despite these limitations the applications are still usually not heavily protected . There's still not the sophistication level we are accustomed to for the Windows platform. As a matter of fact, even if the platform would allow it, there are not much commercial protection frameworks like Armadillo or AsProtect (as far as I know, or if there, are so well hidden that I have not found them, meaning that their market is very little).

The games Symbian scene is a little different indeed (due to N-Gage it has also a greater market)¹⁰. Games are usually written optimizing their performances and in some cases using assembler. When developers have such a great knowledge of the system it's usual that they can also build custom strong protections. It's a usual phenomenon: the first things that start its natural hardening process are the games, then the other software.

I think anyway that the situation is changing. Since I started to collect things for this tutorial I saw a growing interest around the Symbian programs. I hope this document will give a push to the Symbian more talented wannabe crackers ☺

There are several good forums around where you can distribute or take applications. I will not mention them here, because I might give them too much attention and someone might not want it. Anyway it's quite easy to find them..

¹⁰ BTW do you know that it's possible to run N-Gage games on any S60 phone, like the 6600? Search around for some hints about.

3. Reversing the first application: SpriteBackup

I think that the moment has come to get down and dirty with a real application. You should know enough to understand the following explanations. The first application is a commercial application that has been included here due to the simple protection used. Moreover when you start playing with your phone the better thing is to start doing regular backups, and this application saved my phone a lot of times.. ;-)

Target: SpriteBackup 1.0

URL: <http://arteam.accessroot.com/tools/symbian/SpriteBackupV10.zip>

Note: if your phone gets messed up and completely blocked use this resetting procedure then restore the backup did by SpriteBackup.

The soft-format code for Series 60 phones is *#7370# . You enter this code as you would enter a phone number in the Phone application. It performs a format of the Internal drive - All data will be lost if you use this feature.

If this fails consider that exists also another more hard reset procedure anyway that's good for all Series 60 phones.

The method of performing a hard-format, i.e resetting it completely to factory defaults and removing all data, is as follows:

1. Switch off the phone.
2. Hold down the following three buttons: Green (call answer) button, * button, and '3' button
3. While holding these buttons, press the power button and switch on the phone
4. When the message 'Formatting' appears on the screen, release all of the buttons

I suggest also to follow this link for other beginners tricks: <http://www.filesaveas.com/6600.html>

The original program has a time limit protection, the usual time trial limit (see Figure 12): after having placed the code the program reports a MessageBox telling that it's wrong (of course). I have a trial period of 10 days.



Figure 12 – Original SpriteBackup limitations

And, really important it has always an English interface (not changing with nationalized phone settings I mean). This means that finding references to resources will be easier.

These are the installed files:

backupimage.mbm
 restoreimage.mbm
 s60settings.dat
 slogo.mbm
 spritebackup.aif
 spritebackup.app
 spritebackup.rsc
 spritebackup_caption.rsc

multiple bitmap, use MBM Wizard to see it
 multiple bitmap, use MBM Wizard to see it
 settings for s60 phones
 multiple bitmap, use MBM Wizard to see it
 aif file
 real application, use IDA ©
 resource file, use RSC Editor or erl
 resource file, use RSC Editor or erl

Fire-up IDA, do the previously said improvements in the analysis done and read after.

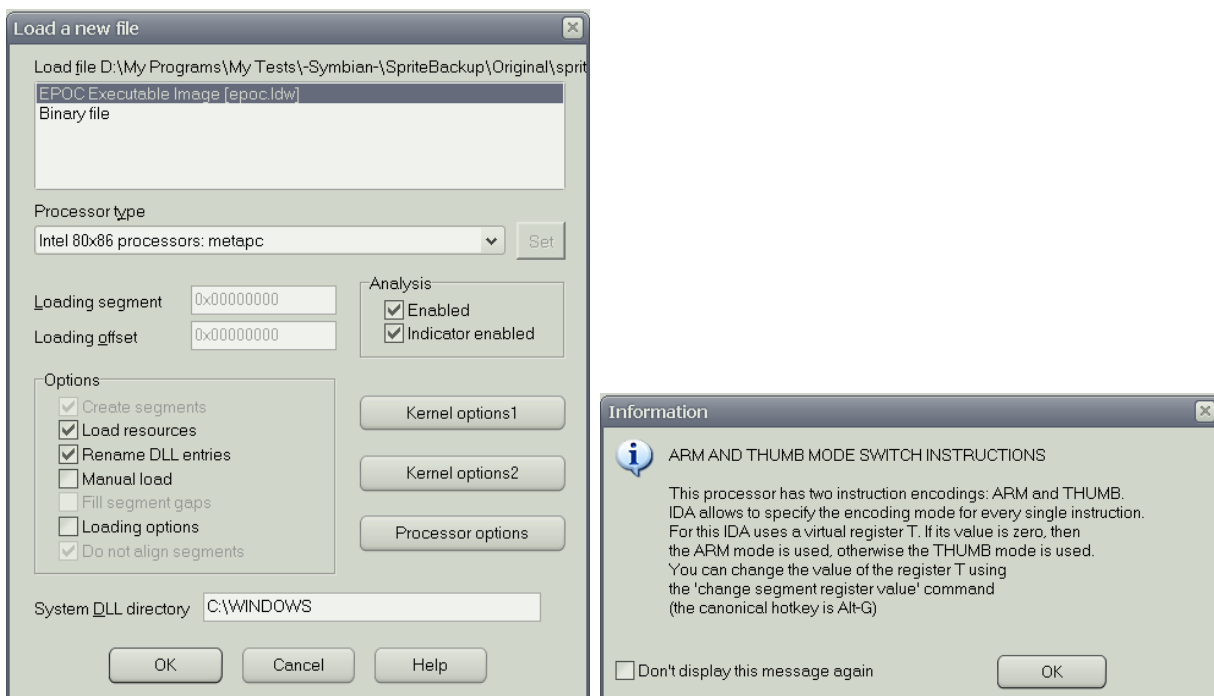
The decompiled resource file is easily got using erl.exe. You can obtain something like the following:

ResNr	ResHex	ResourceID	ResType	Resource	(Note: ResType is only a suggestion!)
1	1h	8F6F001	Unknown	''	
2	2h	8F6F002	String	Sprite Backup	
3	3h	8F6F003	HotKeys		
4	4h	8F6F004	HotKeys	Ge	
5	5h	8F6F005	Defaults	'' '' ''	
6	6h	8F6F006	Buttons	P^OptionsGeExit	
7	7h	8F6F007	Buttons	P^Options;eBack	
8	8h	8F6F008	Buttons	PStart;eBack	
9	9h	8F6F009	Buttons	PStart;eBack	
10	Ah	8F6F00A	Buttons	P^SelecteCancel	
11	Bh	8F6F00B	Buttons	P^YesNo	
12	Ch	8F6F00C	Buttons	P^OKeCancel	
13	Dh	8F6F00D	Buttons	P^OKeBack	
14	Eh	8F6F00E	Buttons	P^OK	
15	Fh	8F6F00F	Buttons	PeCancel	
16	10h	8F6F010	Buttons	P-eDone	
17	11h	8F6F011	Menu (22) *	'' ''	
18	12h	8F6F012	Unknown	eBackup OptionseRestore Optionseselect Restore	

FileeAbouteHelp

3.1. How to crack this nut

Load the app into IDA and tell it to also load the resources, then select Ok to the warning eventually might appear (as following)



I must first make aware of all the unfortunalties due to how the strings are references with Symbian; IDA is not able to identify all the strings into the program.

We follow the approach at § 2.5.5 and find that there are several interesting strings into the program. The most interesting (I will skip the failed guess) is the "Invalid Code in registry". Search for it using a Hex-Editor and you will see that it is referenced at the offset 0x10410.

Go into IDA and use the menu: Jump -> Jump to File Offset, you should land at the address 0x10010390.

If you followed the procedure already described before for strings (§2.5.5), IDA should have recognized some strings (those in Unicode Wide Pascal format), but in the other cases you will find something like the following: a blob of not recognized data.

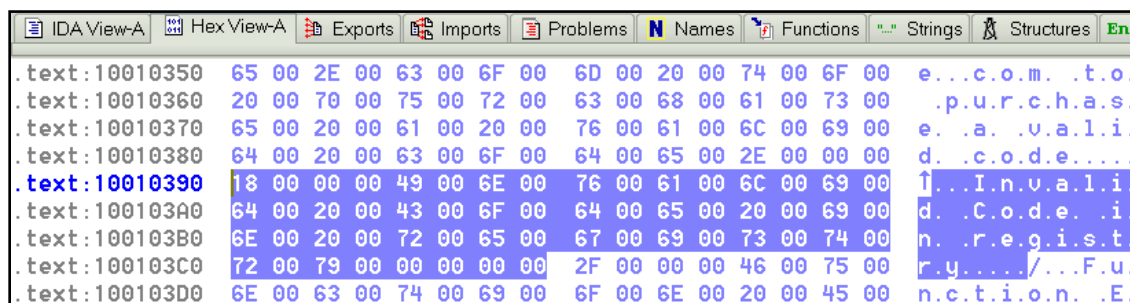
Remember not all the strings are in the Unicode Wide Pascal format, some are just simple Unicode, so if you select one default strings format for an application you might fail finding the other.

```

text:100102B8      unicode 0, <t www.spritesoftware.com to purchase a valid code.>,0
text:10010390 dword_10010390 DCD 0x18, 0x6E0049, 0x610076, 0x69006C, 0x200064, 0x6F0043
text:10010390      ; DATA XREF: start:off_10005A00fo
text:10010390      DCD 0x650064, 0x690020, 0x20006E, 0x650072, 0x690067, 0x740073
text:10010390      DCD 0x790072, 0

```

If you go into the Hex-View-A tag, you should see this:



Take care to have this view synchronized with IDA-View-A.

Note: with IDA to correctly find the opcode of the current instruction it is better to place the cursor in the middle of the operands (in the IDA-View-A) and not in the middle of the ASM instruction.

For example:

```

MOV    R2, SP      -> place the cursor in the operands and not in the "MOV"
BL     loc_100015A4 -> place the cursor in the operands and not in the "BL"

```

The following is a generic procedure you can follow all the times a string is not recognized by IDA.

What you have to do is undefine the currently defined dword_10010390 label, pressing 'U' or using the contextual menu.

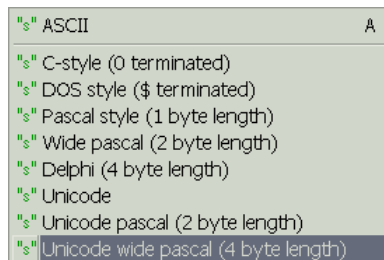
You should get something like the following:

```

.text:10010390 unk_10010390 DCB 0x10 ; DATA XREF: start-off_1000580
.text:10010391 DCB 0
.text:10010392 DCB 0
.text:10010393 DCB 0
.text:10010394 DCB 0x49 ; I
.text:10010395 DCB 0
.text:10010396 DCB 0x6E ; n
.text:10010397 DCB 0
.text:10010398 DCB 0x76 ; u
.text:10010399 DCB 0
.text:1001039A DCB 0x61 ; a
.text:1001039B DCB 0
.text:1001039C DCB 0x6C ; l
.text:1001039D DCB 0
.text:1001039E DCB 0x69 ; i
.text:1001039F DCB 0
.text:100103A0 DCB 0x64 ; d
.text:100103A1 DCB 0
.text:100103A2 DCB 0x20
.text:100103A3 DCB 0
.text:100103A4 DCB 0x43 ; C
.text:100103A5 DCB 0
.text:100103A6 DCB 0x6F ; o
.text:100103A7 DCB 0
.text:100103A8 DCB 0x64 ; d
.text:100103A9 DCB 0
.text:100103AA DCB 0x65 ; e
.text:100103AB DCB 0
.text:100103AC DCB 0x20
.text:100103AD DCB 0
.text:100103AE DCB 0x69 ; i
.text:100103AF DCB 0
.text:100103B0 DCB 0x6E ; n
.text:100103B1 DCB 0
.text:100103B2 DCB 0x20
.text:100103B3 DCB 0
.text:100103B4 DCB 0x72 ; r
.text:100103B5 DCB 0
.text:100103B6 DCB 0x65 ; e
.text:100103B7 DCB 0
.text:100103B8 DCB 0x67 ; g
.text:100103B9 DCB 0
.text:100103BA DCB 0x69 ; i
.text:100103BB DCB 0
.text:100103BC DCB 0x73 ; s
.text:100103BD DCB 0
.text:100103BE DCB 0x74 ; t
.text:100103BF DCB 0
.text:100103C0 DCB 0x72 ; r
.text:100103C1 DCB 0
.text:100103C2 DCB 0x79 ; y
.text:100103C3 DCB 0
.text:100103C4 DCB 0
.text:100103C5 DCB 0


```

Then with the cursor on the "unk_10010390" label use the string definition button as following:



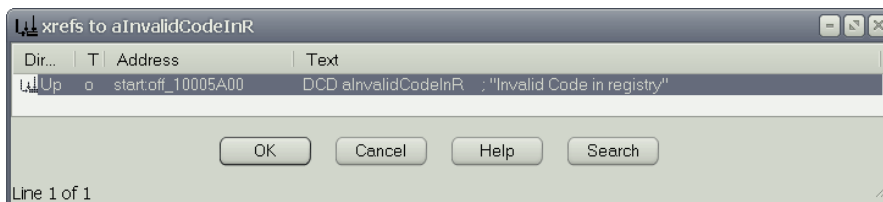
You should get this result:


```
.text:10010390 aInvalidCodeInR DCD 24 ; DATA XREF: start:off_10005A00To
.text:10010390 unicode 0, <Invalid Code in registry>
.text:100103C4 ALIGN 8
```

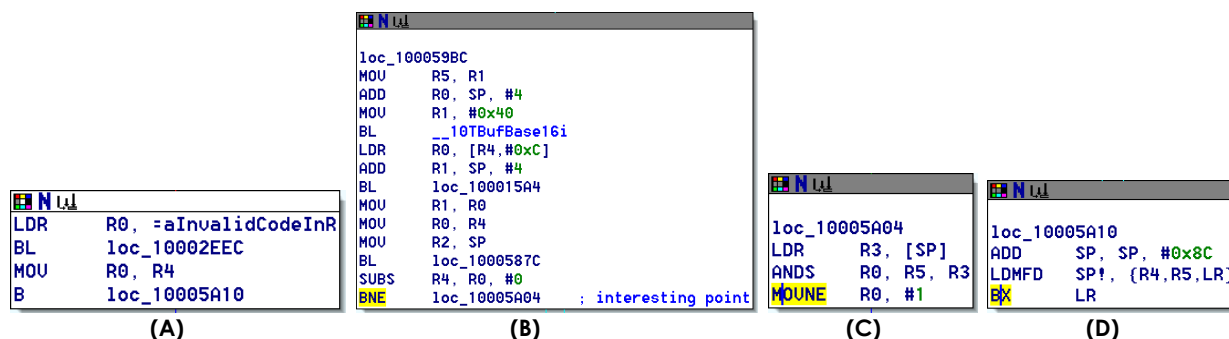
If you selected the default string format in the options as Unicode Wide Pascal, you can simply press the strings button .

Note: Because of the nature how the descriptors (strings) work there is no need to 0 terminating them in most cases. The 0 are usually added by the compiler align to 32 bit boundary.

Now use the 'X' referencing hot-key to see where the string is referenced:



It is referenced at (A) which is called by (B). The branch at the end of (B) lands in (C) and go forward to (D) which then exits from the function.

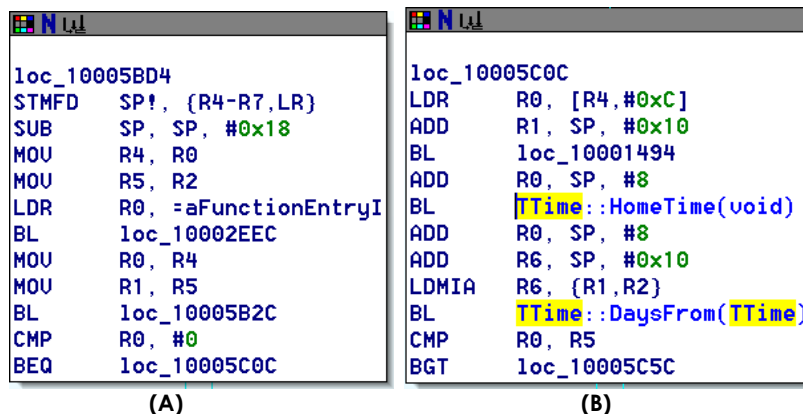


Patch: The idea is to transform the BNE loc_10005A04 into a BE loc_10005A04.

Another alternative is to avoid the time expiry we saw in Figure 12. Just few instructions after the point where the string aInvalidCodeInR is defined, there's another interesting string (I have already redefined):

```
.text:1001047C aFunctionEntryI unicode 0, <#>,0 ; DATA XREF: start:off_10005C08To
.text:1001047C unicode 0, <Function Entry: IsEvaluationExpired>,0
```

This string is referenced at off_10005C08 (A) in the following capture, which calls (B) if the test is false.



The two BL branches in (B) point to calls to some TTime methods, which function is clearly self explanatory; the final BGT means "Branch if Greater Than".

3.1.1 Using desquirr to confirm the guess and find other ways to peel the cat

Just as an exercise, because things are already clear without using it, I also tried to give the code we identified to desquirr to see if there's something interesting I lost.

As a result desquirr gives the result under here which I elaborated a bit just after..

```

100059bc  loc_100059BC:
100059bc      R5 = R1;
100059c8      __10TBufBase16i(& 4, 0x40, R2, R3);
100059d4      R0 = loc_100015A4(* (R4 + 12), & 4, R2, R3);
100059e4      R0 = loc2_1000587C(R4, R0, SP, R3);
100059e8      Cond = R0 - 0;
100059e8      R4 = R0 - 0;
100059ec      if (Cond != 0)
                    goto loc_10005A04;
100059f4      FileLogger(L"Invalid Code in registry", R1, R2, R3);
100059f8      R0 = R4;
100059fc      goto loc_10005A10;

10005a04  loc_10005A04:
10005a04      R3 = * SP;
10005a08      Cond = R5 & R3;
10005a08      R0 = R5 & R3;
10005a0c      if (Cond == 0)
                    goto loc_10005A10;
10005a0c      R0 = 1;

10005a10  loc_10005A10:
10005a14      /* pop */
10005a14      /* pop */
10005a14      /* pop */
10005a18      return R0;

```

Doing some logical elaborations of the code this is the result:

```

100059bc  loc_100059BC:
100059bc      R5 = R1;
100059c8      __10TBufBase16i(& 4, 0x40, R2, R3);
100059d4      R0 = loc_100015A4(* (R4 + 12), & 4, R2, R3);
100059e4      R0 = loc_1000587C(R4, R0, SP, R3);
100059e8      R4 = R0 - 0;
100059ec      if (R4 != 0) { /* our patch just inverts this condition */
                    R3 = * SP; /*10005a04*/
                    R0 = R5 & R3;
                    if (R0 == 0)
                        return R0; /*10005a10*/
                    R0 = 1; /*10005a0c*/
                }
100059f4      FileLogger(L"Invalid Code in registry", R1, R2, R3);
100059f8      R0 = R4;
10005a10      return R0;

```

For the code above you can also find others nice ways to patch the program. Our change of the conditional branch at the address 100059ec forces the program to execute the "if" body. But you can do the same in different ways:

- you can change the condition of the "if" statement, this means change the value of R4 to 0 with a MOV R4,0 at address 100059e8 in order to change the if result.
- if you carefully look the code at 100059e8 you would see:


```

R4 = R0 - 0;
if (R4 != 0)
    ...

```

This code can also be written as:

```

if (R0 != 0)
    ...

```

then setting R0=0 would do the work too: the value of R0 is set by the "call" at loc_1000587C which if you look at it sets R0 at the location 10006048¹¹.

- If the "if" gets executed it sets R0=1 at location 10005a0c, if not R0 is set to a different value at 100059f8, specifically R0=R4 (R4 is also used as test condition), then you can also change here the instruction at 100059f8 into a MOV R0, 1.

3.1.2 Patching the target and testing

We are going now to do the patch, see if the opcode we used is correct then upload to the phone and test. Finally we will pack a new sis file to distribute.

The opcode of the instructions is the following¹²:

IDA Address	File Offset ¹³	Opcode	Hex value
100059EC	5A68	BNE loc_10005A04	04 00 00 1A

In order to understand how to modify the opcodes we need two documents: the ARM Architecture Reference Manual [23] or the simpler reassume [24]. Also document [19] is useful for this step.

Let's do a deep step-by-step reversing of the opcode. Just because it's the first time..

the BNE is Branch with condition NE. According to reference document [23] or tutorial [24], it is coded as in Figure 13 (see § 4.1.5). The conditions are defined in the condition field on page A3-5.

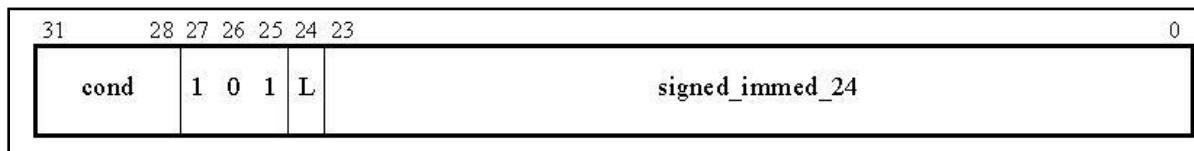


Figure 13 – bits meaning for Branch opcodes

Being a normal Branch (not BL) the L bit is placed to 0, and being the condition NE the cond bits are 0001 (note also the reverse order of bits from 0 to 31).

```
bin: |0...                               ...32|
      |----signed 24 offset----|cond|-B-|L|
      |00000100000000000000000000|0001|101|0|
hex: 04 00 00 1A
```

We want just to change the branch condition so the EQ cond bits are 0000 and the new bits mask becomes:

```
bin: |0...                               ...32|
      |----signed 24 offset----|cond|-B-|L|
      |00000100000000000000000000|0000|101|0|
hex: 04 00 00 0A
```

then, according to our coding, the patch is the following one:

IDA Address	File Offset	Opcode	Hex value
100059EC	5A68	BEQ loc_10005A04	04 00 00 0A

Reassuming now you know that the patch to do is the following:

IDA Address	File Offset	Original Bytes	Modified Bytes
100059EC	5A68	04 00 00 1A	04 00 00 0A

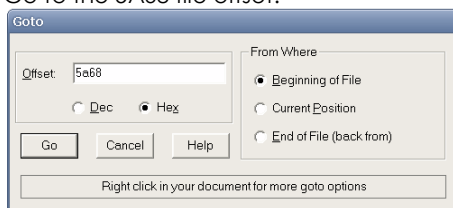
To do it you must use an external Hex Editor, like for example Hex Workshop[42] and open the SpriteBackup.app file

¹¹ We will discover later that loc_1000587C is an interesting point for IMEI calculation..

¹² The file offset is shown in the lower part of IDA interface, in the status line.

¹³ The file offset can be seen in the lower left of the IDA interface, into the position bar

1. Go to the 5A68 file offset:



2. Modify the bytes as we identified:

```
1 50E2 0400 000A
```

3. Save the new app file over the original one

We are not ready to test the application on the real device¹⁴. Just copy the patched application using for example Total Commander and tool [5]. Launch it and this is what you get.



A little note is worth mentioning on how the "signed 24 offset" value has been calculated. In this example we didn't need to change it, but generally speaking it might be needed to patch the application.

The document [23] reports that this offset is calculated as following:

Specifies the address to branch to. The branch target address is calculated by:

1. Sign-extending the 24-bit signed (two's complement) immediate to 32 bits.
2. Shifting the result left two bits.
3. Adding this to the contents of the PC, which contains the address of the branch instruction plus 8.

This is the situation of the code around the target:

.text:100059EC	BNE	loc_10005A04	; interesting point
.text:100059F0	LDR	R0, =aInvalidCodeInR	
.text:100059F4	BL	loc_10002EEC	
.text:100059F8	MOU	R0, R4	
.text:100059FC	B	loc_10005A10	
.text:100059FC	-----		
.text:10005A00	off_10005A00	DCD	aInvalidCodeInR ; DATA XREF: start+59F0↑r
.text:10005A00			; "\x18Invalid Code in registry"
.text:10005A04	-----		
.text:10005A04			
.text:10005A04	loc_10005A04	LDR	R3, [SP] ; CODE XREF: start+59EC↑j

Follow calculations (take care or where the bit 0 is). According to documentation the used formula is:

$$PC = PC + (\text{SignExtend}(\text{signed_immed_24}) \ll 2)$$

¹⁴ Generally speaking before launching you might want to follow a more conservative approach, opening with IDA again the patched application to see if the new opcodes are exactly what you wanted.

The original offset used was:

```
0..      ..24
00000100000000000000000000
```

Revert the bytes order so as to be able to use it with the Windows's calc program:

```
0..      ..24
00000000000000000000000100000
```

Converted to hex is: 0x20

PC at the branch is: 0x100059EC

Add to 0x100059EC the value 0x20 and subtract 0x8 (as documented) and you get:

$\text{Jump_destination} = \text{PC} + \text{off} - 0x8 = 0x100059EC + 0x20 - 0x8 = 0x10005A04$

If you revert the formula just used you will get a generic a algorithm you can use to calculate the opcode of the branch:

1. Calculate offset_24:
 $\text{Offset_24} = \text{jmp_dest} - \text{PC} + 0x8 = 0x10005A04 - 0x100059EC + 0x8 = 0x20$
2. Align bits of the Offset_24 to 24 bits:

```
0..      ..24
00000000000000000000000100000
```
3. Revert the bits order from MSB to LSB notation

```
0..      ..24
00000100000000000000000000
```
4. Convert in hex format and align to 3 bytes:

```
04 00 00
```

3.1.3 Creating a new SIS

Now that we successfully patched the application we have to create a new SIS file (not for distribution of course ;-)). We will use MakeSIS [8].

Decompress the SIS file into one folder, for example like in Figure 14 and follow the steps shown.

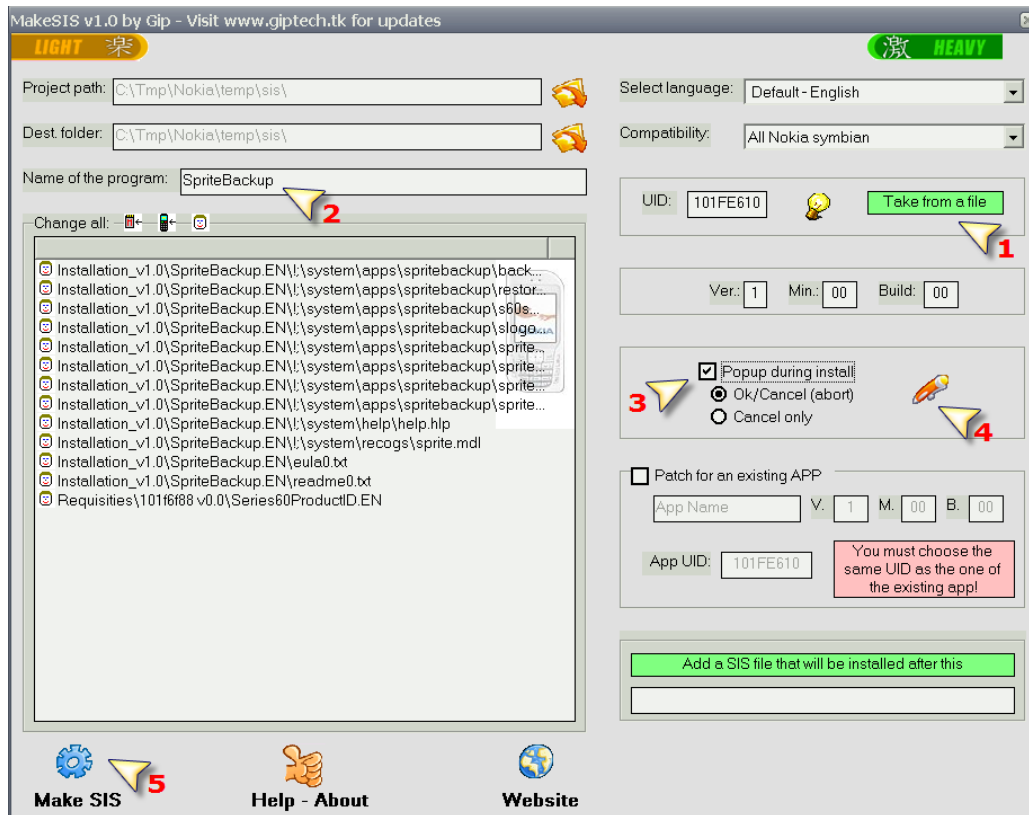
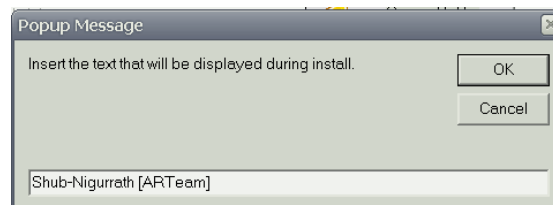


Figure 14 – MakeSIS settings

At step 4 insert you message:



And finally you have a new patched SIS file.

4. Reversing the second application: CoolMMS

Time has come for another application. This time I will work on CoolMMS a nice application that allows you to create your own customized picture to be sent through an MMS. Whether you like it or not doesn't matter, what matters here is its protection ☺

Target: CoolMMS V 1.0.6

URL: http://arteam.accessroot.com/tools/symbian/CoolMMS_S60_V106.sis

Install the application and try to explore it, just to see how it behaves. With reference to Figure 15, it accepts an unlock key (A,B,C) and if not registered reports it in the received MMS (D).



Figure 15 – Original CoolMMS limitations

The installed files are the following:

```
Images\
Templates\
CoolMMS.aif
CoolMMS.app
CoolMMS.idb
CoolMMS.mbm
CoolMMS.r01
CoolMMS.app.bsdat
CoolMMS_caption.r01
```

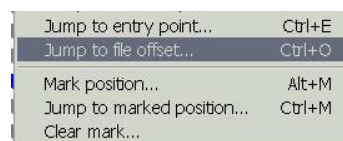
4.1. How to Crack this nut

For this application we will see how to patch it and also how to reverse an existing patch, using EBDS [25]. Also this program is linked with the strings used and doesn't completely rely on external resources. This simplifies things because you can start from the bad-boy message and return backward to the point where the good and bad boy messages tracks splits.

First of all we will use the command Strings from Sysinternals, already introduced before.

There's an interesting string [UNREGISTERED], which is what the program inserts into the MMS subject when the application is not registered. Now we can open the Hex-Editor and search which is the offset where it is stored.

The offset is 0x2EFE6.



Time then for opening IDA, analyze the application with all the instruments and settings introduced before and then go to the offset found using "jump to offset" menu.

You land in the following point.

Below picture is the result of undefining the string identified by IDA, and just after redefining it (as did with SpriteBackup before): the strings were not recognized by IDA. I then renamed the names IDA assigns to the strings with two more explanatory names. You can change the name of a label assigned by IDA pressing 'N'.

```

.text:1002EF04 COOL_Unregistered DCD 0x17, 0x6F0043, 0x6C006F, 0x4D0020, 0x53004D, 0x5B0020
.text:1002EF04 ; DATA XREF: sub_1001FC78:off_1001FCCcTo
.text:1002EF04 DCD 0x4E0055, 0x450052, 0x490047, 0x540053, 0x520045, 0x440045
.text:1002EF04 DCD 0x5D
.text:1002EF38 COOL_Registered DCD 8, 0x6F0043, 0x6C006F, 0x4D0020, 0x53004D, 0
.text:1002EF38 ; DATA XREF: sub_1001FC78:off_1001FD5Cto

```

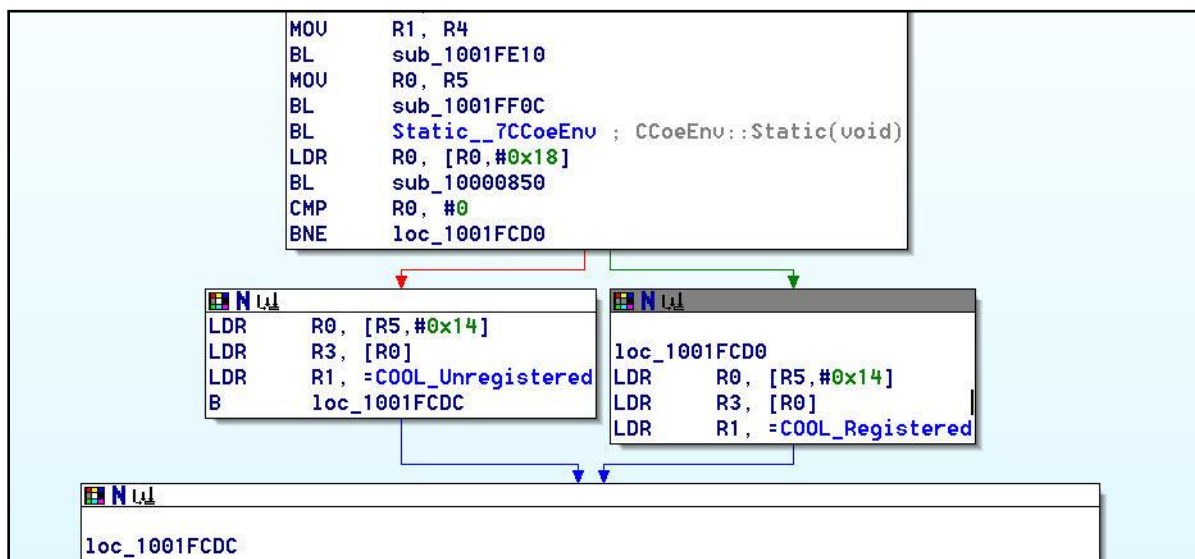
If you see the Hex-View-A you can find the string we were searching for. You should note that the string has an identifier before the first character (0x17) which has been correctly identified by IDA.

```

.text:1002EF00 64 00 00 00 17 00 00 00 43 00 6F 00 6F 00 6C 00 d...C.o.o.l.
.text:1002EF10 20 00 4D 00 4D 00 53 00 20 00 5B 00 55 00 4E 00 .M.M.S. [U.N.
.text:1002EF20 52 00 45 00 47 00 49 00 53 00 54 00 45 00 52 00 R.E.G.I.S.T.E.R.
.text:1002EF30 45 00 44 00 5D 00 00 00 08 00 00 00 43 00 6F 00 E.D.]...C.o.
.text:1002EF40 6F 00 6C 00 20 00 4D 00 4D 00 53 00 00 00 00 00 o.l. .M.M.S....

```

If you press 'X' to find the references of the string you will land here¹⁵:



A clear branch, but despite branch the sub_10000850 doesn't tell much (at least to me).

Well, let's try another way and at the end we will see how the two are connected..

Go back to the results of the Strings batch file. We will look more closely to the strings inside the program and we find these two:

```

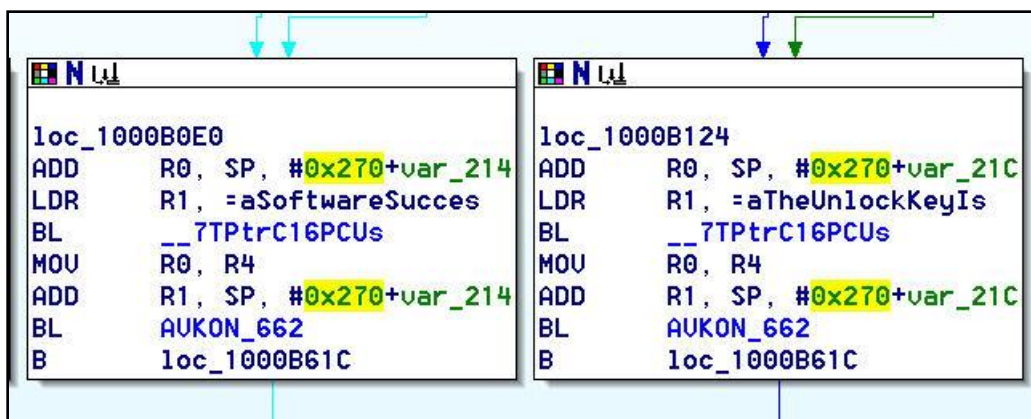
"The unlock Key is incorrect"
"Software Successfully Licensed"

```

Which are definitely interesting!

As before: search the offset with an hex-editor, find it into IDA with jump to offset, redefine the strings and rename if needed then press 'X' to find who is using them (seems a lot of work but once you get accustomed it's very fast). You land here (Note: I did the snapshots when I was still without the additional IDS, so you see a reference for example to AVKON_662):

¹⁵ If you are using IDA 5.x and you see the old plain text format use the contextual menu and select "Graph View"



Back-tracing the execution tracks you will land at the point where the good and bad boys track splits:

```

loc_1000B038
ADD    R0, SP, #0x270+var_54
MOU    R1, #0xE
BL     __9TBufBase8i
BL     NewLC__25CCnvCharacterSetConverter ; CCnvCharacterSetConverter::NewLC(void)
MOU    R6, R0
ADD    R4, SP, #0x270+var_204
MOU    R3, #0
STR    R3, [SP, #0x270+var_204]
MOU    R0, R4
MOU    R1, #4
BL     Connect__3RFsi ; RFs::Connect(int)
MOU    R0, R6
LDR    R1, =(loc_10004CC4+2)
MOU    R2, R4
BL     PrepareToConvertToOrFromLC__25CCnvCharacterSetConverterUiR3RFs ; CCnvCharacterSetConverter::PrepareToConvertToOrFromL(uint, RFs &)
ADD    R0, SP, #0x270+var_3C
BL     Ptr__C7TDesC16 ; TDesC16::Ptr(void)
MOU    R1, R0
ADD    R0, SP, #0x270+var_20C
BL     __7TPtrC16PCUs
ADD    R5, SP, #0x270+var_54
MOU    R0, R6
MOU    R1, R5
ADD    R2, SP, #0x270+var_20C
BL     ConvertFromUnicode__C25CCnvCharacterSetConverterR5TDes8RC7TDesC16 ; CCnvCharacterSetConverter::ConvertFromUnicode(TDes8 &, TDesC16 const &)
MOU    R0, R4
BL     Close__11RHandleBase ; RHandleBase::Close(void)
BL     PopAndDestroy__12CleanupStack ; CleanupStack::PopAndDestroy(void)
MOU    R0, R7
BL     AUKON_42
MOU    R1, R5
BL     sub_10000858
CMP    R0, #0
BEQ    loc_1000B104

```

The most interesting point is of course the final point:

```

BL     sub_10000858
CMP    R0, #0
BEQ    loc_1000B104

```

If you do the jump the program will land into the bad boy message, then what is needed is to have R0!=0

R0!=0 -> valid key.

Now, of course what we do next is to follow the call to the sub_10000858. It contains some interesting things as we will soon discover.

With Figure 16, suppose that R0!=0 and follow the relative execution path. If R0!=0 the function do a call to sub_10020F90.

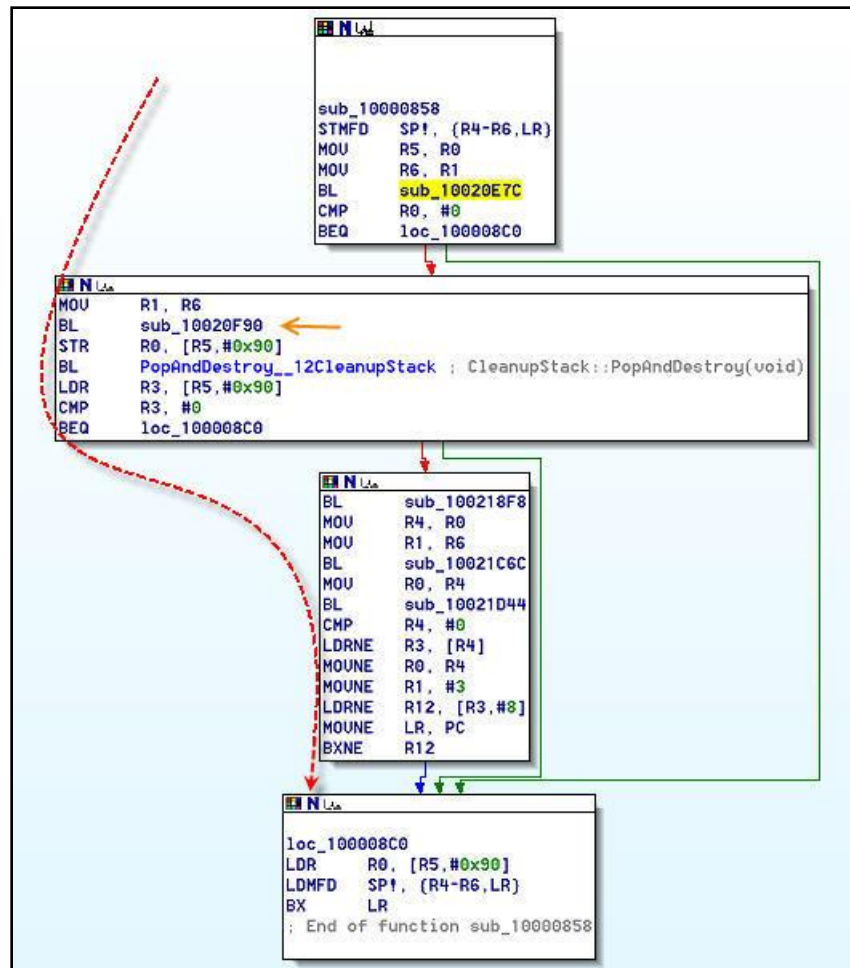



Figure 16 - function sub_10000858

The next step is then to follow the sub_10020F90.

Carefully looking at the disassembly view of Figure 17 we can immediately see that it is the end of our searches, at the end a condition forces R0 to become 1.

What we still need is to check from who this function is used and for what. We can use the function that displays the char of xrefs to the calling functions, using the button .

The result is shown in Figure 18. Well it seems like this function is the only responsible for accepting the key and also seems like the check is made into independent branches.

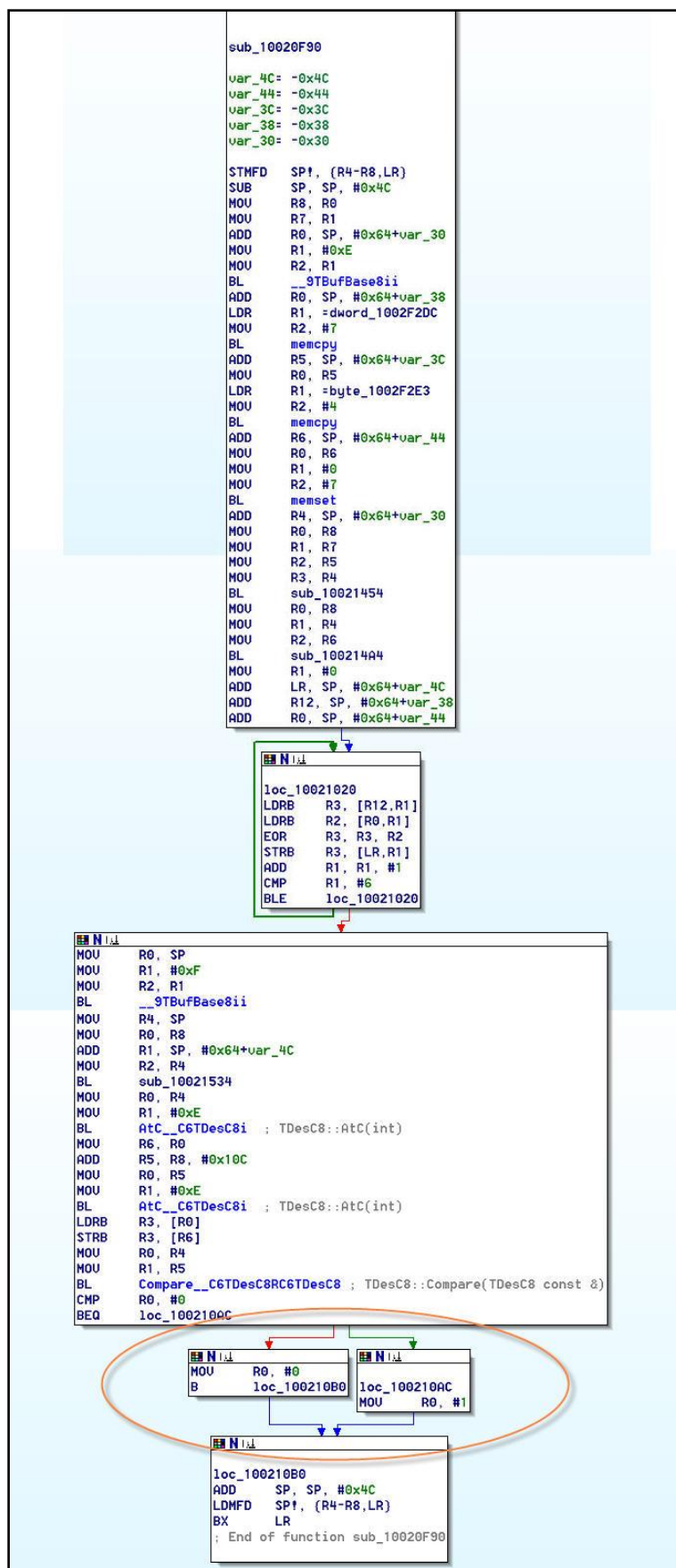


Figure 17 - function sub_10020F90

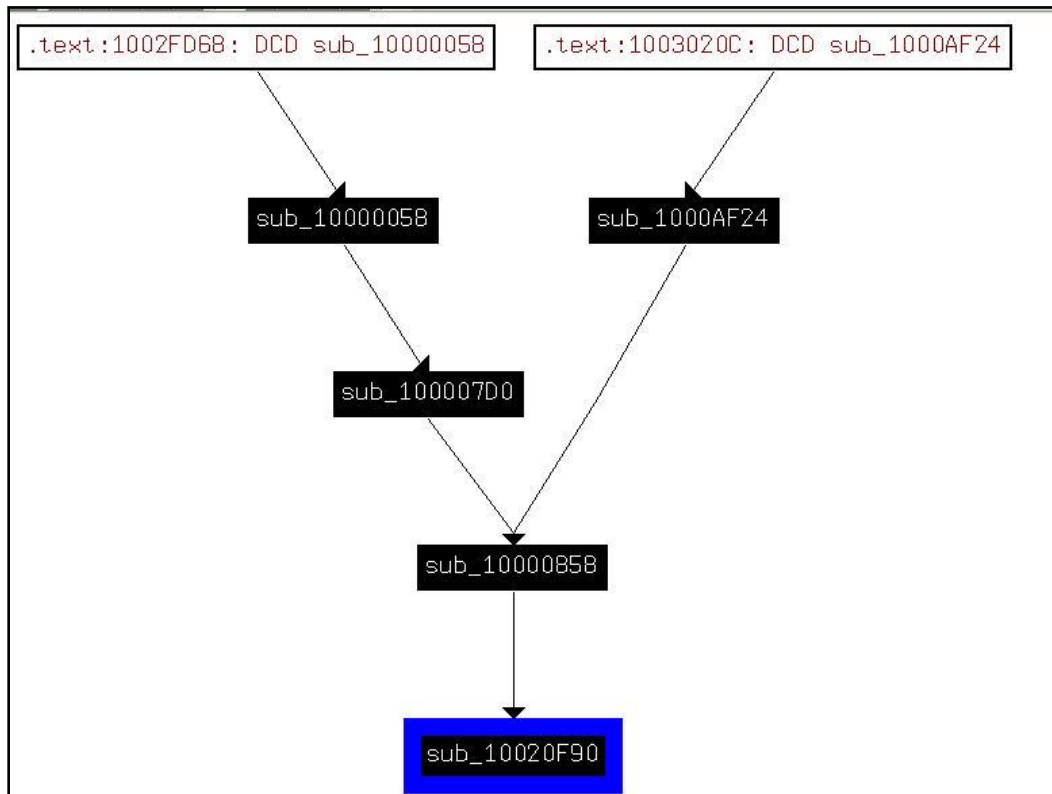


Figure 18 - xrefs chart to sub_10020F90

4.1.1 Creating the first patch

As did for the SpriteBackup we are now changing a branch. We will change the branch:

```
BEQ      loc_100210AC
```

Into a:

```
B      loc_100210AC
```

Another option could be to change the MOV.

The BEQ is Branch with condition EQ.

```

hex: 03 00 00 0A
bin: |0...                               ...32|
      |----signed 24 offset----|cond|-B-|L|
      |00000001100000000000000000|0000|101|0|

```

The values is taken from IDA and converted to 32 bits. IDA can show the opcode of the instruction under the cursor, so take advantage of this functionality, without becoming mad with processor specifications!

Being a normal Branch (not BL) the L bit is placed to 0, and being the condition EQ the cond bits are 0000.

We want to transform the branch to become a simple B, when the condition is omitted the AL condition is used: AL=1110.

so the new binary value is:

```

bin: |0...                               ...32|
      |----signed 24 offset----|cond|-B-|L|
      |00000001100000000000000000|1110|101|0|
new: 03 00 00 EA

```


Fortunately we do not have to recalculate the branch offset and thus the patch is extremely simple!

So the patch becomes:

```
ASM
    old: BEQ      loc_100210AC
    new: B        loc_100210AC
bytecode:
    old: 03 00 00 0A
    new: 03 00 00 EA
unique search pattern:
    03 00 00 EA 00 00 A0
```

4.1.2 Trying out the first patch and find the other..

Patch the application and run it on the phone. After a few playing with it you will discover that not all the limitations have been removed.

To find the way (I will save you all the failed tests) the way is to follow the function called into the sub_10020F90 (the function patched). If you analyze carefully the Figure 17, you should note the following piece of code, just before the first patch we did:

```
BL      __imp_Compare__C6TDesC8RC6TDesC8 ; TDesC8::Compare(TDesC8 const &)
CMP     R0, #0
BEQ     loc_100210AC
```

The call to `TDesC8::Compare(TDesC8 const &)` is interesting. It is this call that definitely decided the following branch, because it modifies R0.

Note: indeed any OS API can modify R0-R3 because those registers are not saved on stack throughout a function call. This is an ARM convention

The Symbian online SDK documentation [26] reports at the following address for Class TDesC8

http://www.symbian.com/developer/techlib/v9.1/docs/doc_source/reference/reference-cpp/N101CA/TDesC8Class.html

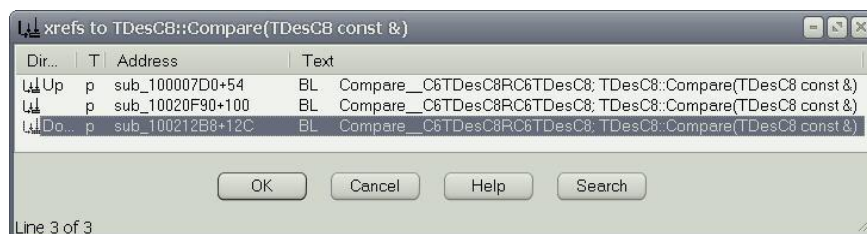
and for the method Compare at the following address:

http://www.symbian.com/developer/techlib/v9.1/docs/doc_source/reference/reference-cpp/N101CA/TDesC8Class.html#%3a%3aTDesC8%3a%3aCompare%28%29

finally reports

Compares this descriptor's data with the specified descriptor's data.

Let see where it is called, using 'X'.



You can check all of them (are only 3) but the one that's most interesting is the one that brings you to the sub_100212B8. The function is patched as in Figure 19: modifying the branch circled there a whole piece of function that is never executed and then the also never executes the MOV at loc_10021424.

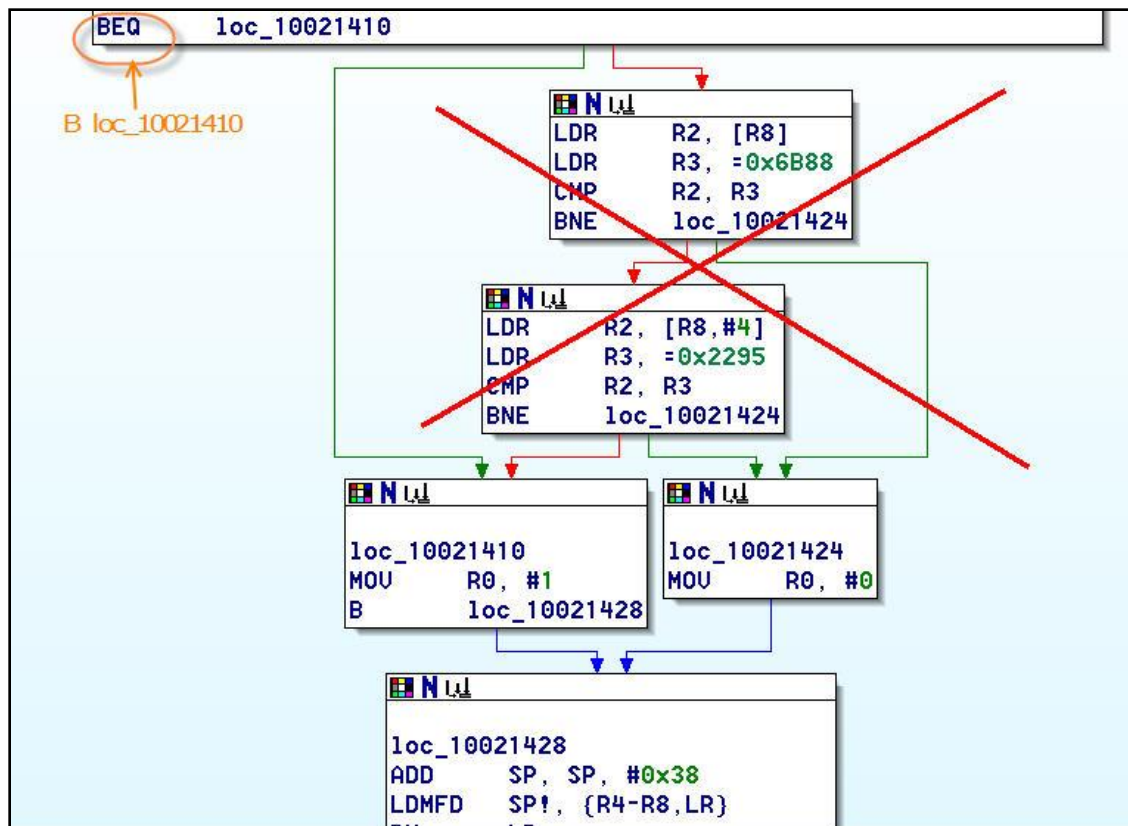


Figure 19 - patched sub_100212B8

Then briefly this is the situation..

```
original ASM:
    BEQ     loc_10021410
original bytecode:
    07 00 00 0A
original binary:
|0...          ...32|
|----signed 24 offset----|cond|-B-|L|
|000001110000000000000000|0000|101|0|
```

Also this time it's enough to change the cond value to 1100

```
new binary:
|0...          ...32|
|----signed 24 offset----|cond|-B-|L|
|000001110000000000000000|1110|101|0|
new bytecode:
    07 00 00 EA
```

the patch number 2 becomes the following:

```
ASM
    old: BEQ     loc_10021410
    new: B       loc_10021410
bytecode:
    old: 07 00 00 0A
    new: 07 00 00 EA
unique search pattern:
    07 00 00 EA 00 20 98
```

4.1.3 Trying out the application with the two patches

Reassuming the patches are the following:

Search Pattern	Original Bytes	Modified Bytes
03 00 00 EA 00 00 A0	03 00 00 0A	03 00 00 EA
07 00 00 EA 00 20 98	07 00 00 0A	07 00 00 EA

This time I chosen to use an unique search pattern to find the place where to patch the application and not the file offset as in Section 3, this approach generally speaking is more conservative because following versions might change the offset where the patch must be done, but not the search pattern. As usual use you Hex Editor to apply it.

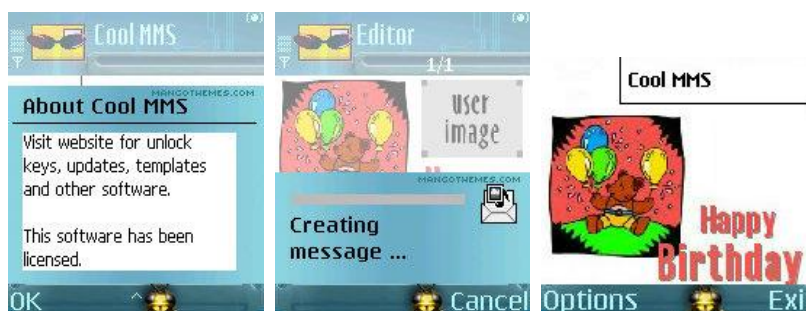
If you now copy the twice patched application to the phone you can test it and see that it works. The only thing that you have to do is to register the application with any serial, long at least 0xD characters. By The way the length check is done in the following piece of code, so if you patch it you will need even a less longer serial:

```

LDR    R3, [SP, #0x270+var_3C]
BIC    R3, R3, #0xF0000000
CMP    R3, #0xD
BGT    loc_1000B038

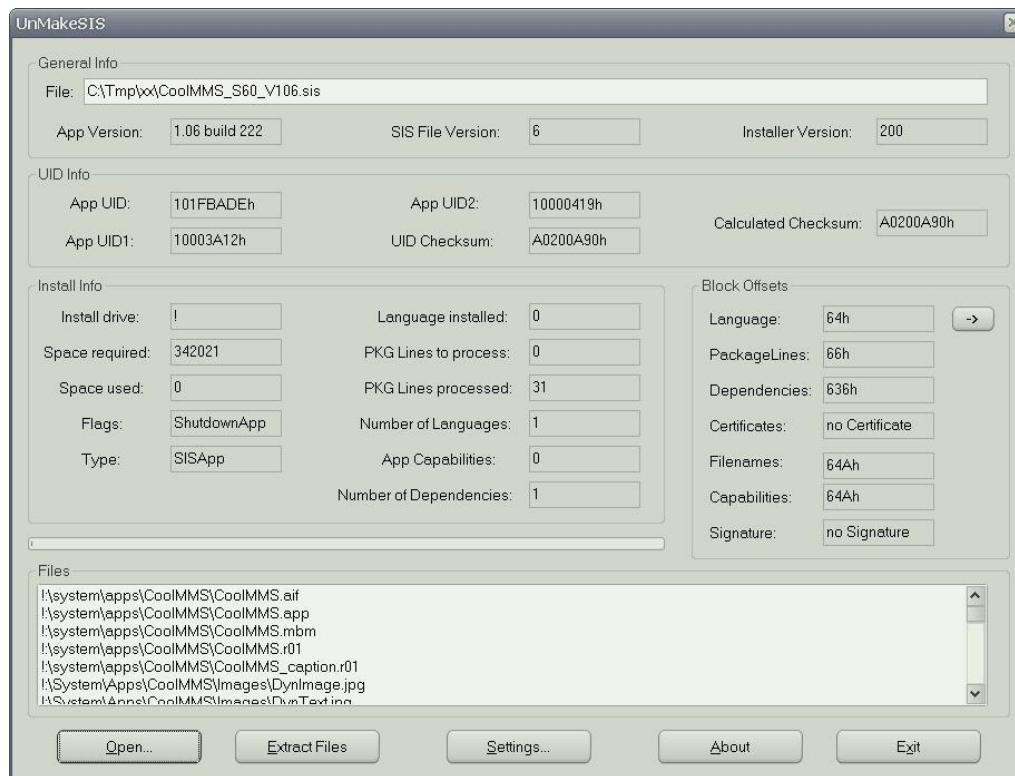
```

Enter the registration number, and you are registered!
Now create the new SIS and you would be ready to go out..

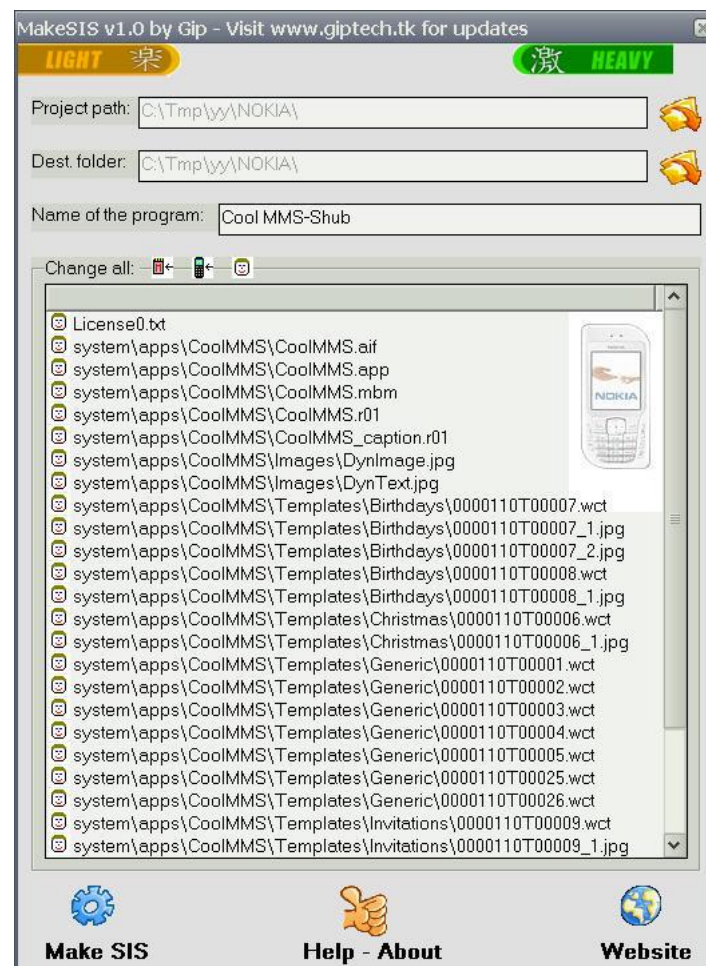


4.2. Creating the new SIS

As done for SpriteBackup we use the tool UnMakeSIS to extract the things:



We can now create a new SIS, once the app has been updated with the patched one:



4.3. Using the EBDS suite

I will just show you what you can obtain with this interesting tool [25]. The first thing to launch is the BDS interface which launches two instances of IDA on the two programs, one original and the second patched.

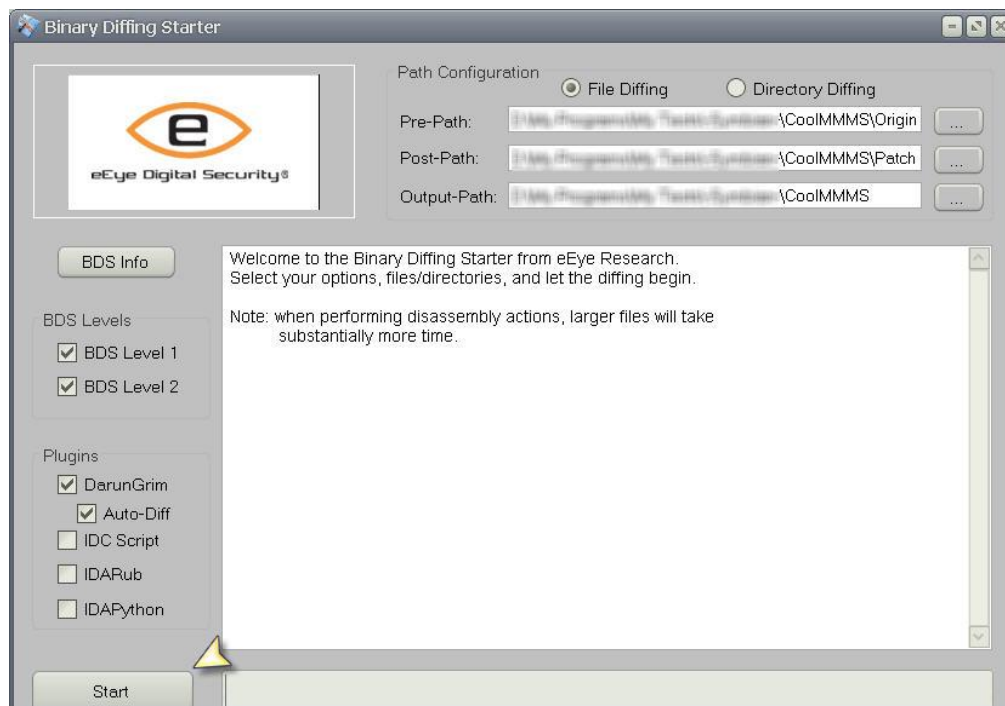
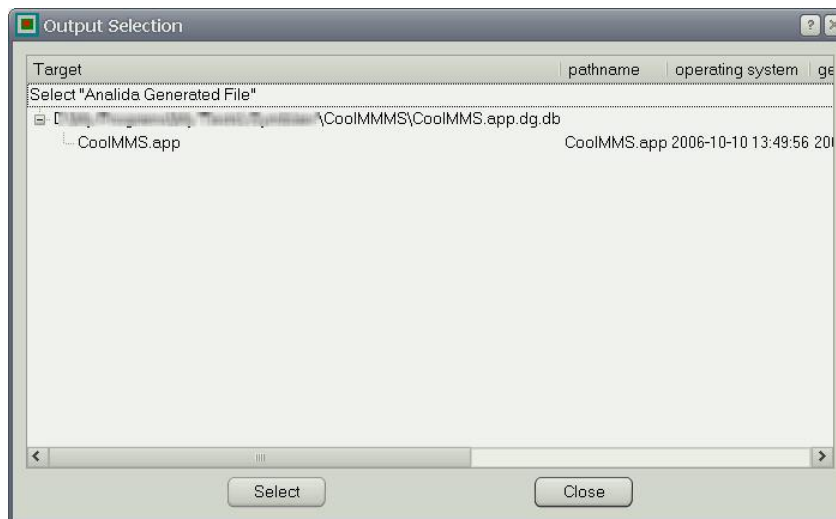


Figure 20 - BDS launch on CoolMMMS

Once you selected the correct files and the correct options (see Figure 20), press Start. The tool launches two instances of IDA on the two files and then creates a unique database of the differences. It creates these two files:

- CoolMMMS.app.dg.db: database of the differences
- CoolMMMS.app.bds: text file which contains the reassume of the just closed analysis. The information reported is especially thought for Win32 exe files, so there's not much interesting things there.

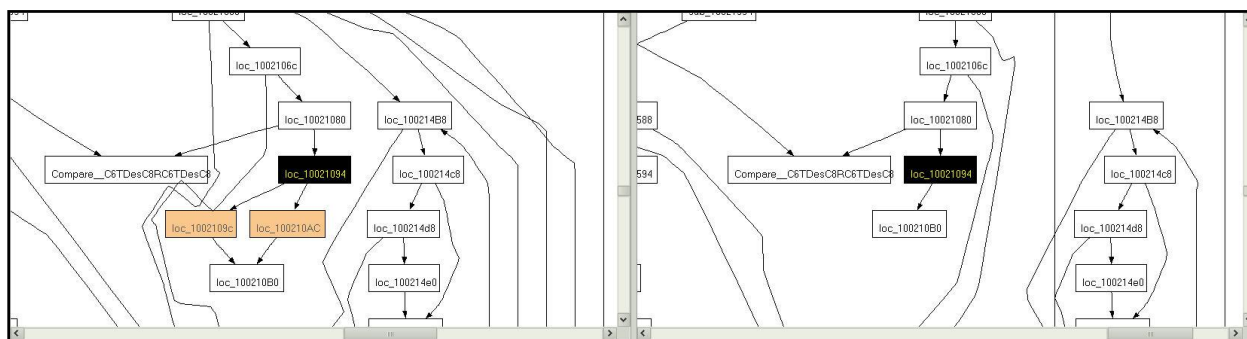
As reported in the tool's documentation, launch DaruGrimmC and select the file just created by BDS.



Open the DaruGrimmC and sort the functions by "Match Rate". If the same subroutine has different assembler instructions (there's a patch) the "Match Rate" decreases: the more it decreases the more are the differences.

Subroutine Match Table					
Subroutine 01	Subroutine 02	Unidentified 01	Unidentified 02	Identified Members	Match Rate
sub_10000058	sub_10000058	2	0	534	0.99753271028
sub_1000AF24	sub_1000AF24	2	0	549	0.9976
sub_1000B934	sub_1000B934	4	0	1842	0.998698481562
sub_1000E8DC	sub_1000E8DC	4	0	2487	0.999035757332
sub_10008980	sub_10008980	4	0	2624	0.999086062452
sub_10003C00	sub_10003C00	4	0	4218	0.999431279621
__11CEikMenuBar	__11CEikMenuBar	0	0	1	1
__17CEikLabelButton	__17CEikLabelButton	0	0	1	1

Use the context menu in order to open the visualizer on the first one (sub_1000058), like in the following picture¹⁶:



What you get is that the same function into the two files differs. Also interesting is the differences view:

original			compare			new			
op	op1	op2	cha	op	op1	op2	op	op1	op2
1	CMP	R0		1	CMP	R0	1	CMP	R0
2	B	loc_100210AC		2	B	loc_100210AC	2	B	loc_100210AC
				3	MOV	R0	3	MOV	R0
						#1			#1

Note that according to the EBDs suite:

Changed column will show whether the line is added or removed with "+/-" sign. "+" sign mean newly added line, "-" sign means removed lines. Also the row is colored with blue or red. Blue colored line is removed line and Red colored line is newly added line. And Gray lines are same lines.

Using IDA the patch we identified with EBDs becomes clear, see Figure 21 and Figure 22.

¹⁶ Using the visualizer is a little tricky because the two views are not synchronized..

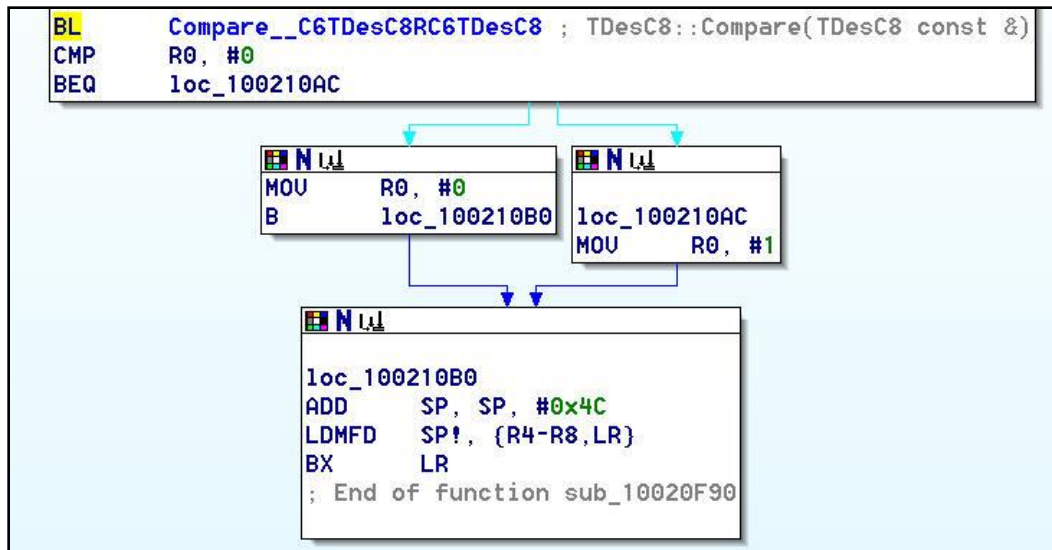


Figure 21 - sub_1000058 original code

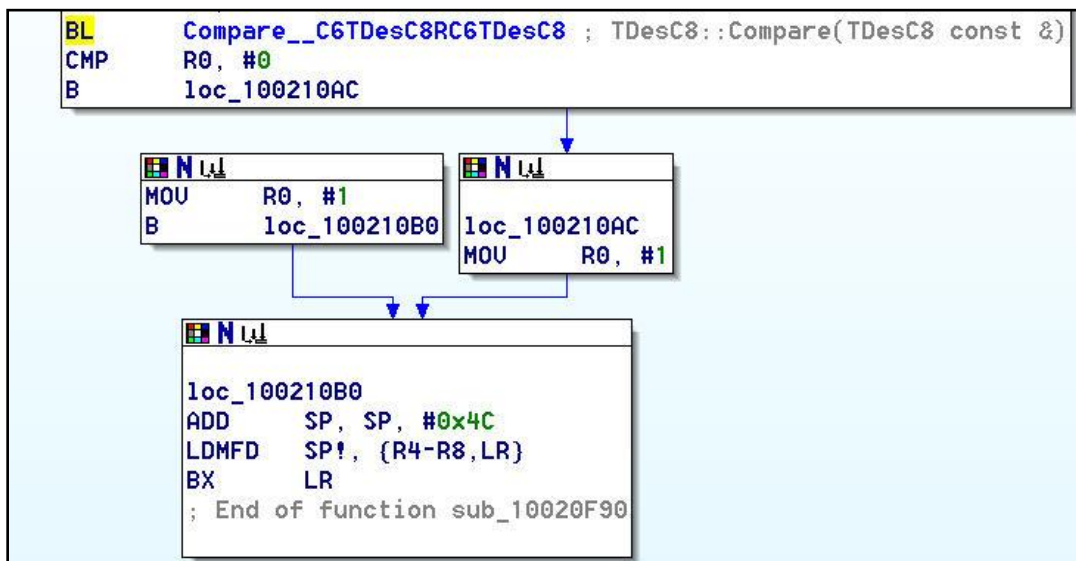


Figure 22 - sub_1000058 patched code

The patch we identified is one we already know.

5. Reversing the third application: MIABO a MIDP 2.0 Java application

This time I decided to work on a little different application, still for the mobile world, but not native. It's a java application, written for the MIDP 2.0 profile (Mobile Information Device Profile for java applications, 2.0, <http://java.sun.com/products/midp/>). MIDP can be considered the Java equivalent of the .NET Compact Framework for PPCs. This profile simply means that the Java Virtual Machine has a standardized set of functionalities the application knows it will be available on compatible devices. Symbian supports it.

Target: MIABO (Message in a Bottle) V 1.1.6

URL: http://arteam.accessroot.com/tools/symbian/MIABO_116.jar or the site <http://www.ugosweb.com/miabo/>

Message in a Bottle is an application for Java Enabled Mobile Phone for exchanging secure encrypted or digitally signed SMS messages using Elliptic Curve Cryptography. With encrypted SMSes Message in a Bottle can be used for sending securely confidential information such as passwords, credit card numbers, phone numbers, etc. and to receive private messages from wherever you are, in order to avoid from password stealing, secrets swiping, industrial espionage and, more in general, SMSes eavesdropping.

With digitally signed SMS, Message in a Bottle can be also used to avoid SMSes from tampering, ensuring integrity and non repudiation. In other words, if someone tries to modify your digitally signed SMS before it arrives to the recipient, the digital signature will be no more valid (because it is computed upon the original SMS) and the recipient can detect the tampering when verifying the SMS.

MIABO has full functionalities for sending up to 20 encrypted or signed SMSes and for receiving unlimited SMSes.



Figure 23 - Original MIABO limitations

I will try to follow a long route to find the correct patch, so as to obtain two results: let you dig a little the application and secondly get used to the logic of this particular way of patching.

Note: the cracking technique used for this MIDP application is the same used for any other Java application on any system, not only Symbian.

5.1. Approach used for the rest of this section

The crack of this application once you understand what to fix is very simple indeed so I thought to follow a different approach, which I think might be more "educational". I will describe three different possible tentative to overcome the limitations of the program, the first two will not be successful, but will help you to understand how to modify the code of a MIDP application (and generally speaking with Java). Only the last approach will be successful. I also myself learnt much more from these different trials than from the last one, which was using indeed not all the possible techniques I used for the other tests. I hope you will follow it, it's not so simple, I know. Many ways to skin a cat, but only one way does it in a good way.

5.2. How to Crack this nut

The application is distributed with a JAR. JAR files are essentially zip archives where the .class compiled java files are stored. What we then need this time is not only IDA, but also a java decompiler to return from class files to java files.

The approach is quite simple and already described in other java cracking tutorials around:

1. decompile the class files back to java files and study them to understand how the program implements the limitations.
2. once you understood the code take IDA and give it the compiled class file. It will take out the standard java Virtual Machine bytecode (the equivalent ASM instructions), almost the same approach already described for example for the .NET world in [20, 21].
3. using the JVM bytecode specifications patch the class file, restore it into the jar file and distribute the new jar.



process.

To accomplish this task we need few tools:

- IDA which is able to disassemble java bytecode
- JAD, an opensource Java Decompiler [27] and a front-end for JAD, like for example my favorite FrontEnd Plus:
<http://www.kpdus.com/frontend/FrontEnd.zip>
- JVM bytecode specifications
- Optionally Total Commander which helps a lot speeding up all the whole

5.2.1 Finding the target on the phone and decompiling it on the PC

Once you have installed the Java application on the phone you have to find it. The MIDP applications are installed under a path like this E:\system\MIDlets\ (if you installed over the smartcard of the phone, otherwise under a folder system\MIDlets anyway).

You can also find the correct path looking here: `system\Apps\[101ce828]\[101ce828].txt` the folders named like [xxxx] are typical of the java applications on Symbian.

If you browse the folder you can see an .app file. What the system creates is a false stub application ([101ce828].app it's not indeed an application) which is used to launch the JVM with the real program's jar file.

MIABO.Jar is available into this path E:\system\MIDlets\[10148f09]\MIABO.jar.

Once the application will be patched or just to test you patches, you will have to upload to the phone the new jar file, and then, using for example FExplorer (see § 2.4) double click on it, in order to re-install the new jar.

Now the first thing to do is to decompile the jar file using **JAD**. You should read the JAD documentation to understand which parameter might be needed for the command-line or use a front-end which worries of everything (like FrontEnd Plus). I chose to use the command-line approach, just because I wanted to understand what's under the hood, and because it perfectly integrates with Total Commander (I can view class files in the lister already decompiled, like native java files ☺).

I stored the jar file into this path: c:\MIABO.

The opened the DOS window and launched this command-line:

```
Jad.exe -o -s java -b -f -ff -a -af -r -safe -space -v -8 -d"c:\MIABO\_jad_output_"
"c:\MIABO\**/*.class"
```

What I got as a result is the folder C:\MIABO_jad_output_ where all the class files were decompiled perfectly. This is the structure you should have under the folder _jad_output_

```
[com]
    [ugos]
        [miabo]
        [midp]
```

*.java, a lot of java files named like a.java, aa.java and so on

There are a lot of java sources indeed but we have to find the correct one! The main program is located under the folder com\ugos\miabo\ which is the namespace the MIABO's author used for his application. Under that folder you will find the file MIABO.java which is our target to analyze.

As usual for these situations, a good start is to look how the application behaves when you insert a fake serial code: I used "123456".

What I can start searching is the string that appears on the phone when I try to enter a registration number. What I found is:

```
case 68: // 'D'
    if (x == null)
    {
        x = new ag("MIABO", "Registration Code", "", 80, 0);
        x.a = this;
        x.b = this;
    }
    ((com.ugos.midp.midlet.WizardMIDlet)this).a(((javax.microedition.lcdui.Displayable)
(x)));
    return;
```

If you carefully read this snippet of java code, you can find that the variable x is an object of type "ag", locally created and it is declared into the file ag.class (or better, the corresponding ag.java).

If you go reading the decompiled ag.java file, you will see that ag is an object tied to the interface of the registration routine, there's no internal logic into the ag class.

Moreover when I enter the fake serial on the program I get the message "Code not valid" then an exception of type "IllegalArgumentException" and with the message "Word not found: 123456" (see Figure 24). The java exceptions are more or less the same as with C++ then I have two options to follow:

1. search the string "Code not valid" and see when/how it is created or not
2. search the string "Word not found"

Apparently it seems better to use the second approach, because the exception interrupts the logical flow of the program and it is much probable that the check is done near the point where the exception has been raised (I expect that the string "Word not found" is an argument of an API raising the exception).



Figure 24 - MIABO exception when registration number is not correct

What I found following the second approach above is this line of code:

```
IllegalArgumentException("Word not found: " + s);
```

It is called into the function `public final byte[] a()` of the file l.class. This is the foreseen exception raising API.

5.3. Where to get JVM bytecode specifications

Before trying to patch the application it is time to find the JVM bytecode specifications in order to be able to do the modifications. Better to explain it here, before understanding what to patch.

The general approach I follow for this application is not to decompile, modify and then recompile the code, but to decompile it, patch the bytecode and not recompile. This saves us from recompilation errors and from recreating the whole compilation environment. The same approach used for [20, 21].

What you must use is Tim Lindholm, Frank Yellin "The Java™ Virtual Machine Specification, Second Edition" [28] and particularly the Chapter 6 "The Java Virtual Machine Instruction Set" (the direct link is <http://java.sun.com/docs/books/vmspec/2nd-edition/html/Instructions.doc.html>). There you can take the instructions bytecode you will soon need.

5.4. Trying out a first patch

The first idea I thought was to force the program not to raise the exception. I know it's not a good idea, because the exception is just the last step of a checking process, but I want to describe it here because it helps in my opinion to understand the way of cracking of java applications.

We already identified the place where the `IllegalArgumentException` is raised: it's inside the file `I.java`

```
if ((i = l.a(s = a.a())) == -1)
{
    throw new IllegalArgumentException("Word not found: " + s);
}
ai[0] = (i & 0x7ff) >>> 3;
ai[1] = (i & 7) << 5;
if ((i = l.a(s = a.a())) == -1)
```

What we want to do is to transform the line

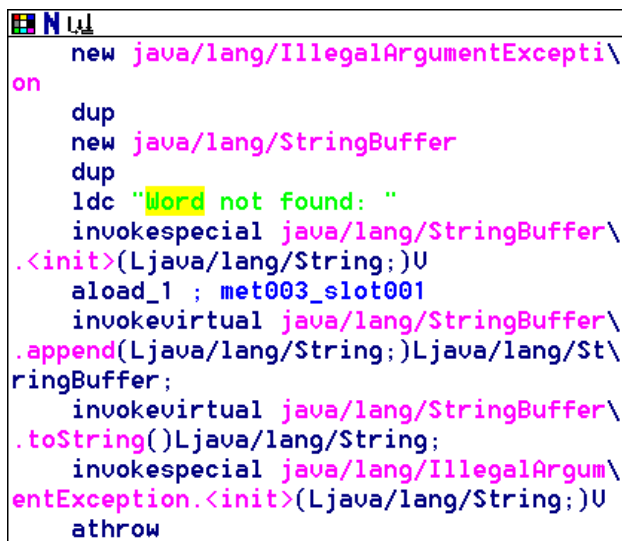
```
if ((i = l.a(s = a.a())) == -1)
```

into

```
if ((i = l.a(s = a.a())) != -1)
```

We now need to properly modify the class file and then what we need is IDA. Open up IDA and drag into the file `I.class`.

Once IDA terminates the initial analysis just use the text search function and search for "Word not found:"



```
new java/lang/IllegalArgumentException\
on
    dup
    new java/lang/StringBuffer
    dup
    ldc "Word not found: "
    invokespecial java/lang/StringBuffer\
    .<init>(Ljava/lang/String;)V
    aload_1 ; met003_slot001
    invokevirtual java/lang/StringBuffer\
    .append(Ljava/lang/String;)Ljava/lang/St\
    ringBuffer;
    invokevirtual java/lang/StringBuffer\
    .toString()Ljava/lang/String;
    invokespecial java/lang/IllegalArgum\
    entException.<init>(Ljava/lang/String;)V
    athrow
```

Before this piece of code (the JVM bytecode equivalent of the "if" body) there's the instruction that owns the direction of jmp we wanted to invert:

```
iconst_m1
if_icmpne met003_59
```

Using the well know synchronization function of the IDA and Hex Views, we can see that the opcode of the if_icmpne is A0 00 1B.

According to bytecode specifications the instruction has this general format:

if_icmpne = A0 op1 op2 branch instruction if int compare not equal ¹⁷

and we want to change it into this second one:

if_icmpeq = branch instruction if int compare is equal ¹⁸

if you follow the links used to find these information (see footnotes) you can see that the documentation is useful just to find if an instruction exists and what it does. It doesn't help to calculate the final opcodes. I must use the java bytecode specifications already introduced into §5.3.

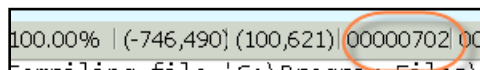
Looking for the instruction if_icmpne you can find this documentation¹⁹ :

```
if_icmp<cond>
branchbyte1
branchbyte2
```

and the documented variations are:

```
if_icmpeq = 159 (0x9F) if_icmpne = 160 (0xA0)
```

then the new bytecode is 9F 00 1B!



Now you can open a Hex-Editor and change A0 00 1B into 9F 00 1B at the given offset (IDA reports it in the status bar).

If you now try supplying JAD the single .class file you will get a new I.java file with contains the modified line:

```
if ((i = l.a(s = a.a())) != -1)
{
    ...
}
```

Note: Generally speaking in order to be sure that you didn't messed anything changing the bytecodes the proof you have to do, before sending all to the phone's JVM, is to decompile the new class with JAD. If JAD is able to decompile the new class file it means that the bytecode is syntactically correct. Moreover you can also double check the modification you did.

Once you changed the I.class reinsert into the original jar file, copy the jar into the phone and then test the application another time... limitations are still present! As already told at the beginning of this section the approach doesn't work, but helped us to introduce the patching procedure and the usage of the JVM bytecode specifications.

¹⁷ See http://jakarta.apache.org/bcel/apidocs/org/apache/bcel/generic/IF_ICMPNE.html

¹⁸ See http://jakarta.apache.org/bcel/apidocs/org/apache/bcel/generic/IF_ICMPEQ.html

¹⁹ See <http://java.sun.com/docs/books/vmspec/2nd-edition/html/Instructions2.doc6.html> and then search for if_icmpne

5.5. Trying out a second patch

As already mentioned before there are two different ways to handle the exception raised (Figure 24). We will search this time for the string "Code not valid".

What we find is this method, into the main MIABO.java file.

```
private void i()
{
    a.a();
    a.a("Registering...", 50);
    try
    {
        l l1 = new l(x.a());
        java.io.ByteArrayOutputStream bytearrayoutputstream = new ByteArrayOutputStream();
        byte abyte0[];
        while ((abyte0 = l1.a()) != null)
        {
            ((java.io.OutputStream) (bytearrayoutputstream)).write(abyte0);
        }
        E = bytearrayoutputstream.toByteArray();
        H();
        ((az) (m)).addElement("1");
        ((bf) (h)).c("encrypted", ah.a(E));
        h.b("PRIVATE", B);
        ((com.ugos.midp.midlet.WizardMIDlet) this).b(69);
        ((com.ugos.midp.midlet.WizardMIDlet) this).e();
        return;
    }
    catch (java.lang.NumberFormatException _ex)
    {
        E = null;
        b("Code has invalid characters");
        return;
    }
    catch (javax.microedition.rms.RecordStoreException recordstoreexception)
    {
        a(((java.lang.Throwable) (recordstoreexception)));
        return;
    }
    catch (java.lang.Exception exception)
    {
        E = null;
        b("Code not valid: " + ((java.lang.Throwable) (exception)).toString());
        return;
    }
}
```

The line of code:

```
b("Code not valid: " + ((java.lang.Throwable) (exception)).toString());
```

is responsible for the message of Figure 24. This exception is part of a catch expression that catches the exception `java.lang.Exception`, raised within the "try" block above. But which of the instructions being part of the "try" body raises the exception?

In the above code the variables are all already created then the first hypothesis is that it might be one of the called methods.

If you remember what the official MIABO documentation told about the limitations you should remember that the author limited the program to 20 SMSes. So the first doubt it came into my mind was "what is the usefulness of placing into some other apparently useful things an `addElement("1");` ?". And this should be the same doubt you should have looking this code for the first time.

Let first of all check if the function `addElement` is able to raise some exception, but we also need the belonging class. The line of code:

```
((az) (m)).addElement("1");
```

Means that the variable `m` is "casted" to the class `(az)` then the `addElement` method is accessed. We have then to look the file `az.java` and see which the parent class is.

The class `az` is declared as:

```
public class az extends java.util.Vector
```

and also redefines the method `addElement` as:

```
public final synchronized void addElement(java.lang.Object obj)
{
    if ((obj instanceof byte[]) || (obj instanceof java.lang.String))
    {
        super.addElement(obj);
        return;
    } else
    {
        throw new IllegalArgumentException("arg must String or byte[]");
    }
}
```

This is simply a stub which checks that the given argument "`obj`" is indeed an object of type "`byte`". Given that this is the situation the line of code executes is the following:

```
super.addElement(obj);
```

The keyword "`super`" means that the method called is the one of the superclass or parent class (`java.util.Vector`). What we have to do is then to search the documentation for `java.util.Vector.addElement`!

We find this documentation²⁰:

addElement

```
public void addElement(Object obj)
```

Adds the specified component to the end of this vector, increasing its size by one. The capacity of this vector is increased if its size becomes greater than its capacity. This method is identical in functionality to the `add(Object)` method (which is part of the `List` interface).

Parameters:

obj - the component to be added.

It doesn't tell anything on exceptions raised by the method, but we will go on this way, because we are explicitly going the wrong way just to test our ability to modify the java source from the bytecode. Even wrong ways to skin the cat are allowed this time!

What we want to try is to substitute the instruction:

```
((az) (m)).addElement("1");
```

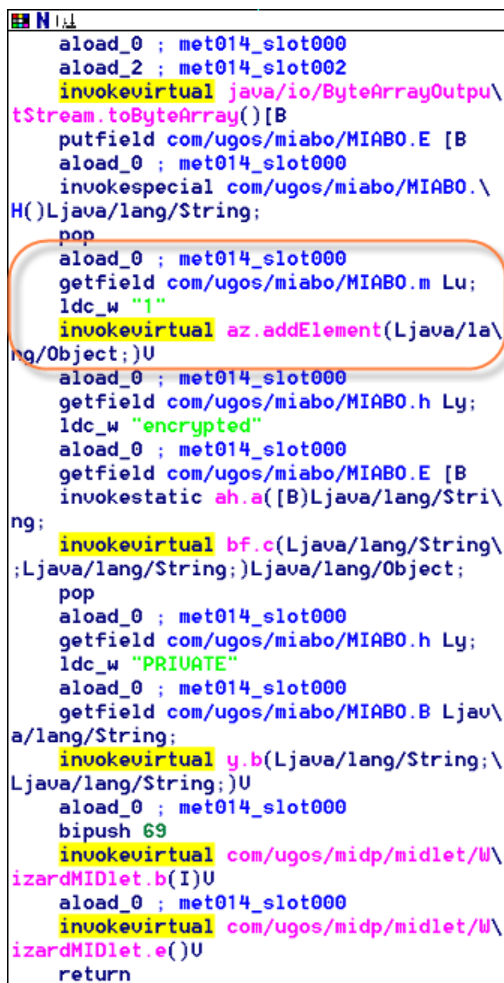
with the following one:

```
((az) (m)).addElement("0");
```

Note that the "`1`" is indeed a string and not a number, then a constant.

Open up IDA and give it the `MIABO.class` file. You can find the same point searching for the neighborhood strings "`PRIVATE`" or "`encrypted`". What you find is:

²⁰ [http://java.sun.com/j2se/1.4.2/docs/api/java/util/Vector.html#addElement\(java.lang.Object\)](http://java.sun.com/j2se/1.4.2/docs/api/java/util/Vector.html#addElement(java.lang.Object))



```

aload_0 ; met014_slot000
aload_2 ; met014_slot002
invokevirtual java/io/ByteArrayOutput\
tStream.toByteArray()[B
putfield com/ugos/miabo/MIAB0.E [B
aload_0 ; met014_slot000
invokespecial com/ugos/miabo/MIAB0.\
H()Ljava/lang/String;
pop
aload_0 ; met014_slot000
getfield com/ugos/miabo/MIAB0.m Lu;
ldc_w "1"
invokevirtual az.addElement(Ljava/la\
ng/Object;)V
aload_0 ; met014_slot000
getfield com/ugos/miabo/MIAB0.h Ly;
ldc_w "encrypted"
aload_0 ; met014_slot000
getfield com/ugos/miabo/MIAB0.E [B
invokestatic ah.a([B)Ljava/lang/Stri\
ng;
invokevirtual bf.c(Ljava/lang/String\
;Ljava/lang/String;)Ljava/lang/Object;
pop
aload_0 ; met014_slot000
getfield com/ugos/miabo/MIAB0.h Ly;
ldc_w "PRIVATE"
aload_0 ; met014_slot000
getfield com/ugos/miabo/MIAB0.B Ljav\
a/lang/String;
invokevirtual y.b(Ljava/lang/String;\
Ljava/lang/String;)U
aload_0 ; met014_slot000
bipush 69
invokevirtual com/ugos/midp/midlet/W\
izardMIDlet.b(I)U
aload_0 ; met014_slot000
invokevirtual com/ugos/midp/midlet/W\
izardMIDlet.e()U
return

```

Figure 25 – where addElement is called

There's a correspondence between the java instruction and the bytecodes, as shown in Figure 25.

Java:

```
((az) (m)).addElement("1");
```

Bytecode:

```

aload_0 ; met014_slot000           // load local variable 0, first declared, it's m
getfield com/ugos/miabo/MIAB0.m Lu; //gets the m variable
ldc_w "1"                          //gets from the constants pool the value "1"
invokevirtual az.addElement(Ljava/lang/Object;)V //calls the addElement function

```

Opcodes:

```

2A          //aload_0 pushed on the stack
B4 00 78    //getfield #0078, popped on the stack the #objectref
13 01 78    //ldc_w constant reference is #0178, pushed on the stack
B6 00 E2    //invoke az.addElement, which is function number 00E2,
           //argument popped from the stack

```

The constants strings are stored into a constants pool, from where they are loaded by reference (as happens with .NET assemblies). This practically is a load by offset, where the offset is an offset from the beginning of a table of constants, a thing already seen with Win32 executables.

Each frame contains an array of variables known as its local variables. The length of the local variable array of a frame is determined at compile time and supplied in the binary representation of a class or interface along with the code for the method associated with the frame.

ldc_w deserves a special attention

```
ldc_w
```

```
indexbyte1
indexbyte2
```

The unsigned indexbyte1 and indexbyte2 are assembled into an unsigned 16-bit index into the runtime constant pool of the current class (§3.6), where the value of the index is calculated as $(\text{indexbyte1} \ll 8) \mid \text{indexbyte2}$. The index must be a valid index into the runtime constant pool of the current class. The runtime constant pool entry at the index either must be a runtime constant of type int or float, or must be a symbolic reference to a string literal (§5.1).

The value 0x178, argument of ldc_w, is the reference to number of string "1".

Unfortunately IDA is not able to explicitly show the constants pools, so we have to find it on our own. To find the correct place where the string is stored we can use a little trick.

1. change the reference number to the reference before, so change with an Hex-Editor 13 01 78 to 13 01 77. This way you are getting the string before in the constants pool of this specific class.
2. patch and give the temporarily modified class file to IDA to see what string you get referenced in the same position.
This time I get this:

```
ldc_w "/com/ugos/miabo/res/sign1.2.png"
```

3. search with the Hex-Editor where this string is located into the class file and just after its end you should find our bellowed "1".

```
6D2F 7567 6F73 2F6D 6961 626F 2F72 6573 2F6D 7367 5F73 6967 5F6F 7065 6E2E 706E 6701 001D 2F63 6F6D | ng..$/com/ugos/miabo/res/msg_sig_open.png.../com
6162 6F2F 7265 732F 716D 6172 6B2E 706E 6701 001F 2F63 6F6D 2F75 676F 732F 6D69 6162 6F2F 7265 732F | /ugos/miabo/res/qmark.png.../com/ugos/miabo/res/
706E 6701 0001 3101 0001 3A01 0008 3C63 6C69 6E69 743E 0100 063C 696E 6974 3E01 0005 3C6E 6577 3E01 | sign1.2.png...|.....<clinit>...<init>...<new>..
6F75 7401 0001 4201 0001 4301 0007 434F 4E54 4143 5401 0004 436F 6465 0100 1B43 6F64 6520 6861 7320 | ..A...About...B...C...CONTACT...Code...Code has
```

What we do is to change it to 0.

```
6F73 2F6D 6961 626F 2F72 6573 2F6D 7367 5F73 6967 5F6F 7065 6E2E 706E 6701 001D 2F63 6F6D | ng..$/com/ugos/miabo/res/msg_sig_open.png.../com
7265 732F 716D 6172 6B2E 706E 6701 001F 2F63 6F6D 2F75 676F 732F 6D69 6162 6F2F 7265 732F | /ugos/miabo/res/qmark.png.../com/ugos/miabo/res/
0001 3001 0001 3A01 0008 3C63 6C69 6E69 743E 0100 063C 696E 6974 3E01 0005 3C6E 6577 3E01 | sign1.2.png...0.....<clinit>...<init>...<new>..
0001 4201 0001 4301 0007 434F 4E54 4143 5401 0004 436F 6465 0100 1B43 6F64 6520 6861 7320 | ..A...About...B...C...CONTACT...Code...Code has
```

Then upload to the phone the new class and see what happens.. of course the limitation is still present, but this patching tentative helped us to understand how to change constants with class files..

Note: Another possibility is to use a nearby string referenced with the instruction ldc_w (in the above snipped of code for example you can use the string "PRIVATE") which is easily located into the class file. Then calculate the position of the string you really want to locate (the "1") by difference between the two references ID. This approach most of the times doesn't work, because the JVM documentation tells that the constants pools are class based then if the second string is not in the same pool this approach doesn't work.

Anyway this time, as you will soon understand, we are more near the final patch and the technique we learnt will be used again..

5.6. Trying out a third patch

This is the good time, the patch we are going to perform now will be the good one, and the techniques learnt before will be a lot useful.

What I use as initial hook is the following portion of string "This is a shareware version and allows", which is part of the message reporting the shareware reminders. Searching among all the different java files I find that the string is referenced into the main MIABO.java file twice:

1. `r.a("This is a shareware version and allows to send max 20 SMSs");`
2. `r.a("This is a shareware version and allows to send max 40 SMSs");`

There are indeed two limitations to the program. We will see which the differences are.

These two messages are respectively called within the functions q() and r().
The two functions are both called by the main execution function of the applet, run():

```

_L6:
a.a();
if (((javax.microedition.lcdui.List) (s)).getSelectedIndex() == 1)
{
    a.a("Signing and sending message...", 20);
    q();
    break; //Loop/switch isn't completed
}
if (((javax.microedition.lcdui.List) (s)).getSelectedIndex() == 0)
{
    a.a("Encrypting and sending message...", 20);
    r();
} else
{
    a.a("sending message...", 20);
    s();
}
break; //Loop/switch isn't completed

```

Do you clearly see that now the meaning of the two different messages is clear: there's a 20 SMSes limitation on Signing and Sending and a 40 SMSes limitation for just Encryption.

I immediately tell you that the function calls to `q()` and `r()` cannot be directly commented because those two functions do things required to the program. What we will do is to avoid the condition that raises the limitation and then the corresponding messages.

5.6.1 Analyzing the function `q()`

Figure 26 reports the original function `q()` with three interesting points.

- A) It's the place where the message is passed to a function of class `r.java` that raises a `IllegalArgumentException`. This piece of code is the catch body of a `java.lang.ArrayIndexOutOfBoundsException` exception. "`ArrayIndexOutOfBoundsException`" is the exception raised by the JVM when an array of elements bypasses its prefixed dimension (which is 20).
- B) Is an interesting place where an element of value "1" is added to an array. We already seen a line of code similar to this for the first patch tested (§5.4).
- C) It is an interesting place where the variable `m` is called with the method `removeElementAt(0)` which removes the first element of the array. Definitely these lines of code look promising.

This time we want to do the things properly, so our supposition actually is that `m` is an Array (we already seen it §5.4) and that its maximum dimension is forced to 20 elements.

If you search how the object `m` is used (use the string search "`m`") you will find this interesting piece of code into the function `t()`

```

a.a("Creating database...");
for (int i1 = 0; i1 < 20; i1++) {
    (az) (m).addElement(((java.lang.Object) (i1)));
}

```

That is exactly what we were supposing. The array is pre-filled of 20 elements with increasing values from 1 to 20 (less important).

```

private void q()
{
    java.lang.String s1 = ((javax.microedition.lcdui.TextBox) (q)).getString();
    m m1 = new m(16501);
    try
    {
        ((java.util.Vector) (m)).removeElementAt(0); C
        m.b("SMS", B);
        bh bhl = m1.a(D, ((ag) (p)).a(), s1);
        try
        {
            H();
            ((az) (m)).addElement("1"); B
            m.b("SMS", B);
        }
        catch (java.lang.Exception _ex) { }
        c(((z) (bhl)));
        ((com.ugos.midp.midlet.WizardMIDlet) this).b(35);
        ((com.ugos.midp.midlet.WizardMIDlet) this).e();
        return;
    }
    catch (java.io.IOException _ex)
    {
        D();
        ((com.ugos.midp.midlet.WizardMIDlet) this).b(41);
        ((com.ugos.midp.midlet.WizardMIDlet) this).e();
        return;
    }
    catch (java.lang.ArrayIndexOutOfBoundsException _ex)
    {
        D();
        r.b("");
        r.a("This is a shareware version and allows to send max 20 SMSs");
        ((com.ugos.midp.midlet.WizardMIDlet) this).b(100);
        ((com.ugos.midp.midlet.WizardMIDlet) this).e();
        return;
    }
    catch (java.lang.Throwable throwable) A
    {
        D();
        a(throwable);
        return;
    }
}

```

Figure 26 - Original function q()

The function r() is absolutely similar to the function q(), especially for the points A, B and C above.


```

try
{
    ((java.util.Vector) (m)).removeElementAt(0);
    m.b("SMS", B);
    q q1 = m1.a(r1, D, ((ag) (p)).a(), s1);
    try
    {
        H();
        ((az) (m)).addElement("1");
        m.b("SMS", B);
    }
    catch (java.lang.Exception _ex) { }
    c(((z) (q1)));
    ((com.ugos.midp.midlet.WizardMIDlet) this).b(35);
    ((com.ugos.midp.midlet.WizardMIDlet) this).e();
    return;
}
catch (java.io.IOException _ex)
{
    D();
    ((com.ugos.midp.midlet.WizardMIDlet) this).b(41);
    ((com.ugos.midp.midlet.WizardMIDlet) this).e();
    return;
}
catch (java.lang.ArrayIndexOutOfBoundsException _ex)
{
    r.b("");
    r.a("This is a shareware version and allows to send max 40 SMSs");
    ((com.ugos.midp.midlet.WizardMIDlet) this).b(100);
    ((com.ugos.midp.midlet.WizardMIDlet) this).e();
    return;
}
}

```

Figure 27 - Fragment of function r()

5.6.2 Creating the patch

What I want to do to patch is to eliminate the calls B and C.

Looking at the JVM bytecode specifications we can see that the NOP bytecode is 0x00²¹. In order to do the patch we first of all need to identify where are the functions q() and r() seen with IDA. Search (with ALT-T) the string "20 SMSs".

Figure 28 shows where you land.

²¹ <http://java.sun.com/docs/books/vmspec/2nd-edition/html/Instructions2.doc10.html>

```

met024_123:
    .stack
    locals Object com/ugos/miabo/MIAB0
    locals Object java/lang/String
    locals Object m
    stack Object java/lang/ArrayIndexOutOfBoundsException
    .end stack
    pop
    aload_0 ; met024_slot000
    invokespecial com/ugos/miabo/MIAB0.\
D()U
    aload_0 ; met024_slot000
    getfield com/ugos/miabo/MIAB0.r Lar;\
    ldc_w ""
    invokevirtual ar.b(Ljava/lang/String\
;)U
    aload_0 ; met024_slot000
    getfield com/ugos/miabo/MIAB0.r Lar;\
    ldc_w "This is a shareware version a\
nd allows to send max 20 SMSs"
    invokevirtual ar.a(Ljava/lang/String\
;)U
    aload_0 ; met024_slot000
    bipush 100
    invokevirtual com/ugos/midp/midlet/W\
izardMIDlet.b(I)U
    aload_0 ; met024_slot000
    invokevirtual com/ugos/midp/midlet/W\
izardMIDlet.e()U
    return

```

Figure 28 - exception of the 20 SMSs limitation

The "met024_123" is not directly linked to any other element in the code, just because is a catch of an exception thrown somewhere else, fortunately the exceptions map is statically compiled within the class file (obviously) and IDA can resolve it. If you press the key 'X' over the label "met024_123" you land here:

```

.catch java/lang/Exception from met024_58 to met024_90 using met024_90
.catch java/io/IOException from met024_19 to met024_107 using met024_107
.catch java/lang/ArrayIndexOutOfBoundsException from met024_19 to \
met024_107 using met024_123
.catch java/lang/Throwable from met024_19 to met024_107 using met024_159

```

Look the order of the exceptions: it is the same order of the consecutive catches (exception chain) in the function q(): java.lang.Exception, java.io.IOException and finally java.lang.ArrayIndexOutOfBoundsException. Then the code we want to search for, reading backward the exceptions chain, is the top method: "met024_58" (see Figure 29).

```

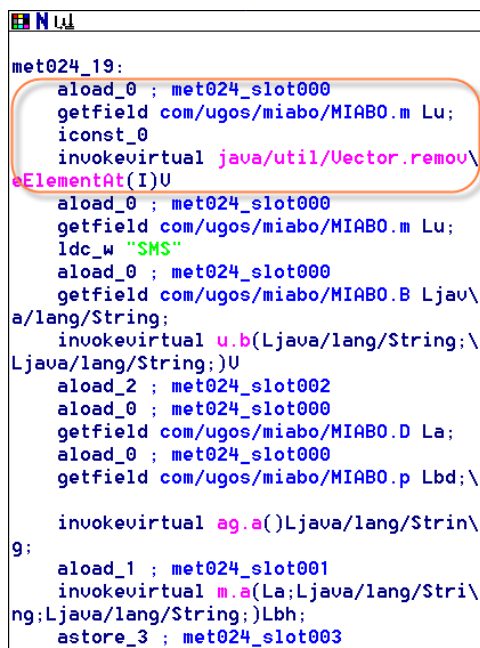
met024_58:
    aload_0 ; met024_slot000
    invokespecial com/ugos/miabo/MIAB0.\
H()Ljava/lang/String;
    pop
    aload_0 ; met024_slot000
    getfield com/ugos/miabo/MIAB0.m Lu;
    ldc_w "1"
    invokevirtual az.addElement(Ljava/la\
ng/Object;)U
    aload_0 ; met024_slot000
    getfield com/ugos/miabo/MIAB0.m Lu;
    ldc_w "SMS"
    aload_0 ; met024_slot000
    getfield com/ugos/miabo/MIAB0.B Ljav\
a/lang/String;
    invokevirtual u.b(Ljava/lang/String;\
Ljava/lang/String;)U
    goto met024_91

```

Figure 29 - met024_58 bytecode view

From the Section §5.5 and Figure 25 you should already know that the addElement call is composed of different instructions. We have to NOP them all.

The same we will do for the `removeElementAt` call, we already identified in the java bytecode: it is just above "met024_58", see Figure 30, it is the "met024_19".



```

met024_19:
    aload_0 ; met024_slot000
    getfield com/ugos/miabo/MIAB0.m Lu;
    iconst_0
    invokevirtual java/util/Vector.removeElementAt(I)U
    aload_0 ; met024_slot000
    getfield com/ugos/miabo/MIAB0.m Lu;
    ldc_w "SMS"
    aload_0 ; met024_slot000
    getfield com/ugos/miabo/MIAB0.B Ljav\
a/lang/String;
    invokevirtual u.b(Ljava/lang/String;\
Ljava/lang/String;)U
    aload_2 ; met024_slot002
    aload_0 ; met024_slot000
    getfield com/ugos/miabo/MIAB0.D La;
    aload_0 ; met024_slot000
    getfield com/ugos/miabo/MIAB0.p Lbd;\

    invokevirtual ag.a()Ljava/lang/Strin\
9;
    aload_1 ; met024_slot001
    invokevirtual m.a(La;Ljava/lang/Stri\
ng;Ljava/lang/String;)Lbh;
    astore_3 ; met024_slot003
  
```

Figure 30 - met024_19 bytecode

How to patch? You should already know, but here are the details in short:

1. with IDA select the instructions we want to NOP and go to the Hex-View-A, the selected bytes are those to NOP.
2A B4 00 78 13 01 78 B6 00 E2
2. open a Hex Editor and patch as following, at the right offset:
old bytes: 2A B4 00 78 13 01 78 B6 00 E2
new bytes: 00 00 00 00 00 00 00 00 00 00

You must do the same for the "met024_19".

The two patched branches "met024_58" and "met024_19" are shown in Figure 31 (taken from the new class file).



```

met024_58:
    aload_0 ; met024_slot000
    invokespecial com/ugos/miabo/MIAB0.\
H()Ljava/lang/String;
    pop
    nop
    nop
    nop
    nop
    nop
    nop
    nop
    nop
    nop
    nop
    nop
    nop
    aload_0 ; met024_slot000
    getfield com/ugos/miabo/MIAB0.m Lu;
    ldc_w "SMS"
    aload_0 ; met024_slot000
    getfield com/ugos/miabo/MIAB0.B Ljav\
a/lang/String;
    invokevirtual u.b(Ljava/lang/String;\
Ljava/lang/String;)U
    goto met024_91

met024_19:
    nop
    nop
    nop
    nop
    nop
    nop
    nop
    nop
    nop
    aload_0 ; met024_slot000
    getfield com/ugos/miabo/MIAB0.m Lu;
    ldc_w "SMS"
    aload_0 ; met024_slot000
    getfield com/ugos/miabo/MIAB0.B Ljav\
a/lang/String;
    invokevirtual u.b(Ljava/lang/String;\
Ljava/lang/String;)U
    aload_2 ; met024_slot002
    aload_0 ; met024_slot000
    getfield com/ugos/miabo/MIAB0.D La;
    aload_0 ; met024_slot000
    getfield com/ugos/miabo/MIAB0.p Lbd;\

    invokevirtual ag.a()Ljava/lang/Strin\
9;
    aload_1 ; met024_slot001
    invokevirtual m.a(La;Ljava/lang/Stri\
ng;Ljava/lang/String;)Lbh;
    astore_3 ; met024_slot003
  
```

Figure 31 - Patched met024_58 and met024_19 of function q()

The same procedure must be done for the function `r()`, but I will leave it for your own experiments.

Now the limitations are really gone and you can use the program over 20 or 40 SMSs.



5.6.3 Changing the About message

Normally I would have terminated here, but it's worth a mention on how to fix the about message, also because have a nice structure. (see Figure 23).

Searching for the string into the java files bring you there:

```
try
{
    java.lang.String s2 = H();
    s1 = "\nVersion: " + ((javax.microedition.midlet.MIDlet) this).getAppProperty("MIDlet-Version") + "\nLicensed to " + s2;
}
catch (java.lang.Exception _ex)
{
    r r1;
    ac ac1;
    byte abyte0[] = (ac1 = (r1 = D.a()).a.c()).c.c();
    boolean flag = false;
    if (abyte0[0] == 0)
    {
        flag = true;
    }
    java.lang.String s3 = "";
    for (int i1 = ((int) (flag)); i1 < abyte0.length; i1 += 8)
    {
        s3 = s3 + l.a(abyte0, i1);
    }
    s1 = "\n\nNot Licensed\nIdentification code:\n" + ah.a((byte) ac1.b()) + " " + s3;
    ((java.util.Hashtable) (h)).remove("encrypted");
}
```

The "Not Licensed" message is handled into a catch of a `java.lang.Exception` exception. Into the corresponding try block the exception is raised by the function `H()` (just because the message just after is now shown, then the execution interrupts into `H()`). You cannot just NOP the call to `H()` because otherwise the program won't run, you loose the reference to the string `s2`, used the line after, and then the program crashes.

You have to patch the lines of code:

```
java.lang.String s2 = H();
s1 = "\nVersion: " + ((javax.microedition.midlet.MIDlet) this).getAppProperty("MIDlet-Version") +
"\nLicensed to " + s2;
```

into:

```
s1 = "\nVersion: " + ((javax.microedition.midlet.MIDlet) this).getAppProperty("MIDlet-Version") +
"\nLicensed to ";
```

Now you have to find where this code is into the byteview of IDA.

But take care, if you try searching for the string "Not Licensed" into IDA and you are in the Graph view you will not find anything (bug?), you must be in the Text View of IDA and then you will land into a place where the string is present (then switch back to Graph View):

```

met016_48:
    aload_0 ; met016_slot000
    invokespecial com/ugos/miabo/MIABO.H()Ljava/lang/String;
    astore_2 ; met016_slot002
    new java/lang/StringBuffer
    dup
    ldc_w "\nVersion:"
    invokespecial java/lang/StringBuffer.<init>(Ljava/lang/String;)U
    aload_0 ; met016_slot000
    ldc_w "MIDlet-Version"
    invokevirtual javax/microedition/midlet/MIDlet.getAppProperty(Ljava/lang/String;)Ljava/lang/String;
    invokevirtual java/lang/StringBuffer.append(Ljava/lang/String;)Ljava/lang/StringBuffer;
    ldc_w "\nLicensed to "
    invokevirtual java/lang/StringBuffer.append(Ljava/lang/String;)Ljava/lang/StringBuffer;
    aload_2 ; met016_slot002
    invokevirtual java/lang/StringBuffer.append(Ljava/lang/String;)Ljava/lang/StringBuffer;
    invokevirtual java/lang/StringBuffer.toString()Ljava/lang/String;
    astore_1 ; met016_slot001
    goto met016_253
  
```

What is interesting are the points A and B, which are the bytecode equivalents of the two instructions we want to modify.

Java

```
java.lang.String s2 = H();
```

Bytecode

```

aload_0 ; met016_slot000
invokespecial com/ugos/miabo/MIABO.H()Ljava/lang/String;
astore_2 ; met016_slot002 //stores the return value as local variable into store slot #2
  
```

The instruction `astore_2` is interesting because, even without looking at the JVM bytecode specifications, it stores the return value of the call (`s2`) to the local variables storing slot number 2. The corresponding instruction is `aload_2`, which stores the local variable #2 into the stack.

The other point to patch is the portion of the instruction that adds the string `s2` to the string `s1`. The adding of string is translated into a call to the method "append". Then this leads us to the patch of point B.

What we need to do is to NOP these two sets of instructions and this is what we will obtain once patched the class file:

```

met016_48:
  nop
  nop
  nop
  nop
  new java/lang/StringBuffer
  dup
  ldc_w "\nVersion:"
  invokespecial java/lang/StringBuffer\
  .<init>(Ljava/lang/String;)U
  aload_0 ; met016_slot000
  ldc_w "MIDlet-Version"
  invokevirtual javax/microedition/mid\
  let/MIDlet.getAppProperty(Ljava/lang/Str\
  ing;)Ljava/lang/String;
  invokevirtual java/lang/StringBuffer\
  .append(Ljava/lang/String;)Ljava/lang/St\
  ringBuffer;
  ldc_w "\nLicensed to "
  invokevirtual java/lang/StringBuffer\
  .append(Ljava/lang/String;)Ljava/lang/St\
  ringBuffer;
  nop
  nop
  nop
  nop
  invokevirtual java/lang/StringBuffer\
  .toString()Ljava/lang/String;
  astore_1 ; met016_slot001
  goto met016_253

```

The new About message is now clean:



6. Some misc Protection Schemes

Inside this section I inserted some things I found for different Symbian programs which are worth mentioning, but are not long enough for a whole chapter.

6.1. Using the IMEI and IMSI numbers for Registration Schemes

The IMEI, short for International Mobile Equipment Identity, is a unique number given to every single mobile phone²². It is not modifiable by the user and a written copy of it can typically be found behind the battery. It can also be displayed on screen by entering the `*#06#` key sequence.

Depending on your mobile, this number can have two different formats (the old and the new format: aabbbb-cc-ddddd-e if your mobile has been manufactured before April 1st 2004. xxxxxxx--ddddd-e if your mobile has been manufactured after this date.

I must forward you to the excellent documents [31, 32, 33, 34] where you can understand everything you need about this subject.

I also suggest reading the tutorials reported at Section §7 which reports some simple cases really interesting.

Anyway it is interesting that most registration number mechanisms are based on the IMEI: they generate a serial number starting from the IMEI or the IMSI numbers. This most of the times allows to easily keygen the applications.

6.1.1 Approaching again SpriteBackup

Once again SpriteBackup, the program already used for Section §3, shows it's educational usefulness. Once you read the tutorials I mentioned before, open SpriteBackup with IDA (the ids file already created the first time). If you check the Imports section you will easily find that the program imports this API:

```
PlpVariant::GetMachineIdL(TBuf<128> &)
```

Which is known to be one of the possible methods offered by the SDK to retrieve the phone's IMEI. Check, through the references, who are calling that API. Here follows the sequence of places where IDA lands you:

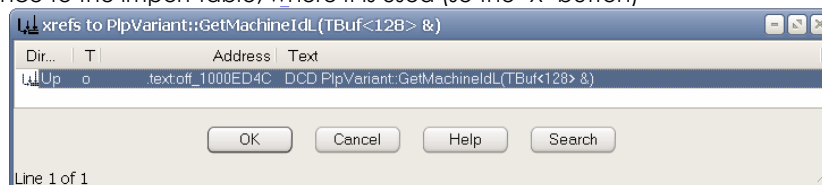
1. the raw import (just use the Imports view)

100127E0	74	GetHelpContext__C11CoeControlR15TCoeHelpContext	CONE
10012D68	6	GetMachineIdL__10PlpVariantRt4TBuf1i128	PLPVARIANT
10012980	303	GetText__C9CEikEdwinR6TDes16	EIKCOCTL

2. the import into the imports table

```
.idata:10012D68 ; Imports from PLPVARIANT[10009b13].DLL
.idata:10012D68 ;
.idata:10012D68 IMPORT _imp_GetMachineIdL__10PlpVariantRt4TBuf1i128
.idata:10012D68 ; DATA XREF: .text:off_1000ED4C↑o
```

3. the reference to the import table, where it is used (se the 'X' button)



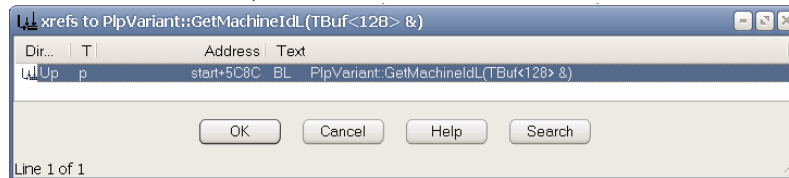
4. the stub created by the compiler (it is by default hidden, to see it press + on keypad)

²² The IMEI identifies the phone itself, not its user nor his subscription. These are identified by the IMSI code. Practically, this means that:

- a user will change of IMEI but not of IMSI when he changes its phone, keeping the same subscription.
- a user will change of IMSI but keep the same IMEI when he changes his subscription, keeping the same phone

```
.text:1000ED40 ; [0000000C BYTES: COLLAPSED FUNCTION P1pVariant::GetMachineIdL(TBuf<128> &). PRESS KEYPAD "+" TO EXPAND]
.text:1000ED4C off_1000ED4C DCD P1pVariant::GetMachineIdL(TBuf<128> &)
.text:1000ED4C ; DATA XREF: P1pVariant::GetMachineIdL(TBuf<128> &)↑r
```

5. the references window to the stub, where it is used



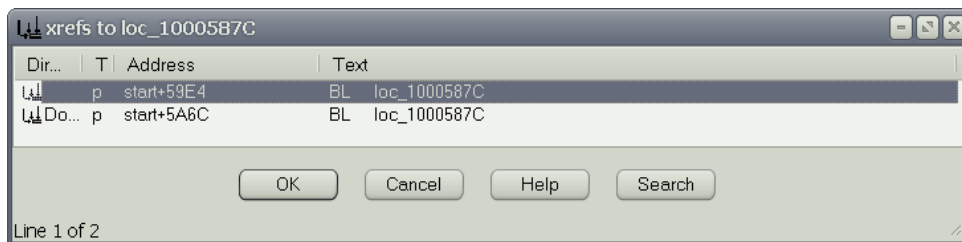
6. the final landing, where actually the import is used (what was before are just compilation issues).

```
loc_10005C6C
STMFD SP!, {R4,R5,LR}
SUB SP, SP, #0x108
MOU R5, R1
MOU R0, SP
MOU R1, #0x80
BL __10TBufBase16i
MOU R4, SP
MOU R0, R4
BL P1pVariant::GetMachineIdL(TBuf<128> &)
MOU R0, R5
MOU R1, R4
BL TDes16::Copy(TDesC16 const &)
LDR R0, =dword_100104C8
BL loc_10002EEC
MOU R0, R5
BL loc_10002EEC
B loc_10005CB4
```

So summarizing if you press (actually several things) 'X' you should land here:

```
callMachineId
STMFD SP!, {R4,R5,LR}
SUB SP, SP, #0x108
MOU R5, R1
MOU R0, SP
MOU R1, #0x80
BL __10TBufBase16i
MOU R4, SP
MOU R0, R4
BL P1pVariant::GetMachineIdL(TBuf<128> &)
MOU R0, R5
MOU R1, R4
BL TDes16::Copy(TDesC16 const &)
LDR R0, =dword_100104C8
BL loc_10002EEC
MOU R0, R5
BL loc_10002EEC
B loc_10005CB4
```

I renamed the location's name as "callMachineId" to improve readability. Once again press the 'X' button to find references and you land at location loc_10005908 which is directly called (no references this time, but a direct link, double click on the incoming blue thin arrow in the Graph View) by loc_1000587C. This last location is called into two positions in the main program:



The first call lands you into the same place already seen in Section §3.1 (figure B)). The second reference brings you to the initial checks the application is doing to see if it's registered (with consequential writing to the log file of the application).

I'll make it brief here, but essentially now we will go patching at a deeper level. If you see the two places where the `loc_1000587C` is called, you will find these two surrounding pieces of code (note that I renamed the `loc_1000587C` into an easier to find `loc2_1000587C`):

```

loc_10005A40
STMFD    SP!, {R4-R7,LR}
SUB      SP, SP, #4
MOU      R5, R0
MOU      R7, R1
MOU      R6, R2
MOU      R4, R3
LDR      R0, =aFunctionEntryCli
BL       loc_10002EEC ; call logging function
MOU      R0, R5
MOU      R1, R6
MOU      R2, SP
BL       loc2_1000587C ; gets IMEI
CMP      R0, #0 ; return condition
BNE      loc_10005A04 ; interesting point

MOU      R2, SP
BL       loc2_1000587C ; gets IMEI
SUBS     R4, R0, #0 ; return condition
BNE      loc_10005A04 ; interesting point

```

The two conditions in A) and B) are interesting: resuming them if `R0!=0` then registration code is good. Go into the `loc2_1000587C` and see what you can do to force that.

Following this approach you again land at the location `loc_10005908`: the interesting part happens after the call to `callMachineId`.

```

BL       callMachineId
MOU      R0, R5
BL       loc_10002EEC
ADD      R0, SP, #8
MOU      R1, #0x14
BL       __9TBufBase8i
ADD      R4, SP, #8
MOU      R0, R4
MOU      R1, R5
BL       TDes8::Copy(TDesC16 const &)
MOU      R0, R4
BL       TDesC8::Ptr(void)
MOU      R1, R0
MOU      R0, R8
MOU      R2, #0xE
BL       loc_10005B80
CMP      R7, R0
MOUNE    R0, #0
MOVEQ    R0, #1
CMP      R6, #0xD50
MOVEQ    R0, #1
CMP      R0, #0
ANDNE    R3, R10, #0xFF
STRNE    R3, [R9]
MOUNE    R0, #1

```

If you notice there are some interesting MOVEQ and MOVNE which are repeatedly playing with register R0 value. What I do now is to change the instruction

```
MOVNE R0, #0
```

into

```
MOVNE R0, #1
```

You should already know at this stage how to do it. If you test this patch, you will see the program now wants a registration number but accepts anything.

Now the situation is much nicer:



6.1.2 Lesson learnt

There are always different ways to peel a cat. First thing before approaching a program is to search for OS called APIs which helps to understand what the application wants from the system. These entry points can then be used to climb the rope back to the correct patching point. Before starting to struggle on the Assembler always do what I call "Bird Fly Reversing" @.

6.2. A simple protection using a complex license manager framework

I decided to add another small application so as to show how to manage with time limited or execution limited applications.

Target: 3D Minigolf

URL: http://arteam.accessroot.com/tools/symbian/Tect_3D_Mini_Golf_v1.35_S60.SIS

The application is protected by a common framework (don't know how much has been updated recently): License Manager Pro version 2.42 (<http://openbit.com>).



Once you launch the application you are prompted with a sequence like the following:



As you can see the common framework LMPRO is called to handle the registration. We will have two options to patch the application: patch the application itself or patch the framework. This time we will patch the application, because the framework is too complex and would require a tutorial on its own.

6.2.1 Patch the application

This first approach is simple because the application has a message reporting that we have "xx days left of trial" or that the trial is over ("This is your last trial use. On the next startup you will be asked to buy a license.").

Of course at this stage I can skip all the explanations on how to correctly find the strings into IDA and directly go to the point that interests us.

If you analyze the 3D_miniGolf.APP into IDA you will easily find that the above messages are built, see Figure 32.

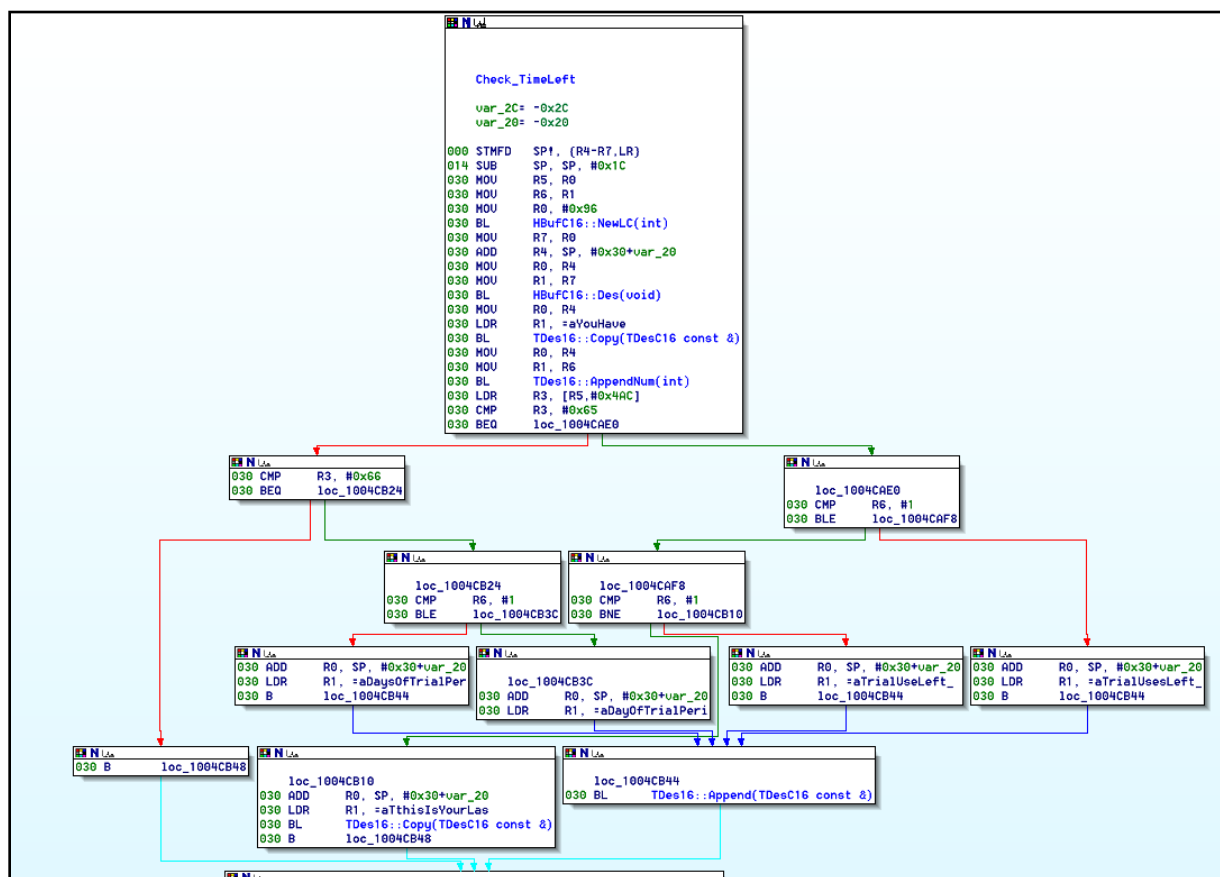
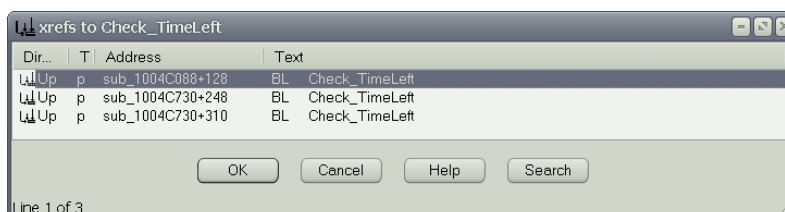
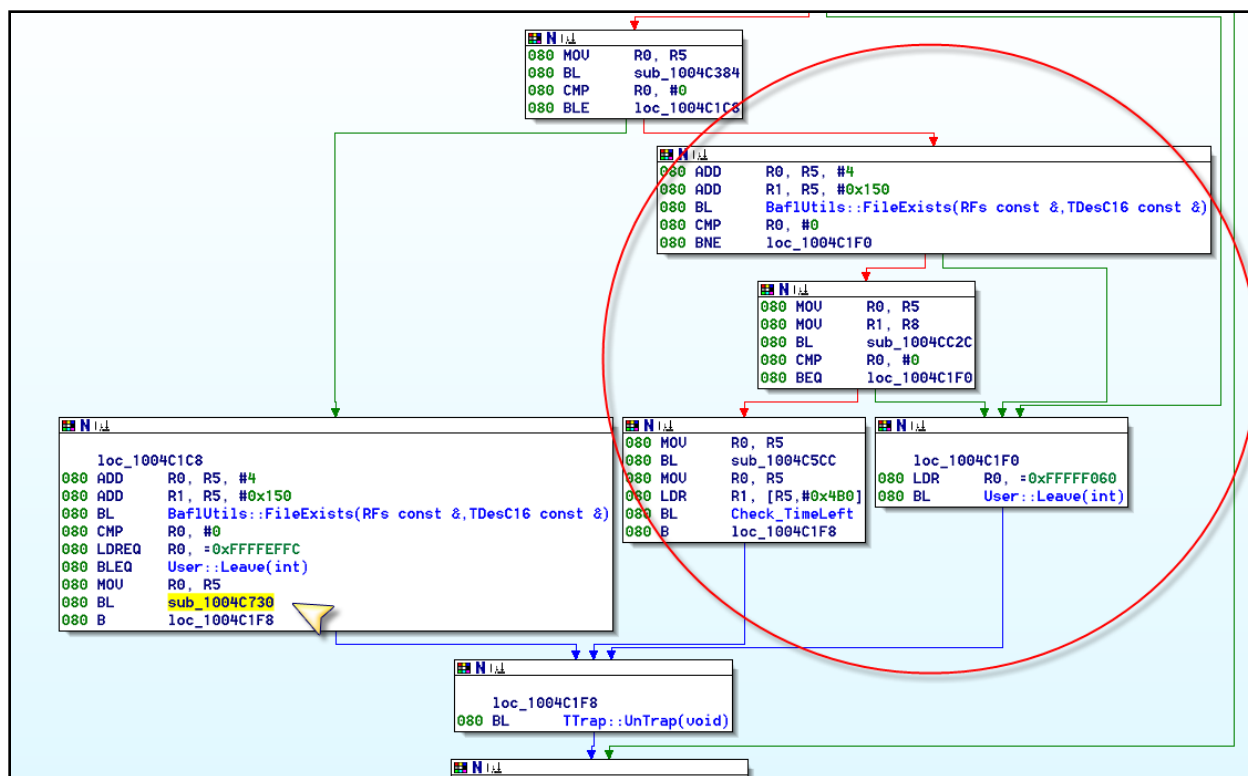


Figure 32 - Function where expiration message is composed

It's a function with a quite obvious structure. Anyway I renamed it as "Check_TimeLeft", so as to recognize it easily in the code. If you search where it is called you will find these results:



Three places which we have to skip all. Let see the first call:



You land into a function that could you take to think that to skip the check it would be enough to patch the “BLE loc_1004C1C8”, where the two branches splits. But, if you see the other xrefs to the function “Check_TimeLeft” you will soon discover that are called by the function sub_1004C730 (see arrow above).

The first think that came to my mind is then to NOP this call. But how can which is the NOP opcode?

As already mentioned in [19, 24]: on ARM “MOV R0, R0” is used as NOP, however it contain nulls so any other “neutral” instruction have to be used when writing proof of concept codes for vulnerabilities, “MOV R1, R1” is just an example.

```
MOV R1, R1 -> 01 10 A0 E123
```

Anyway MOV R0, R0 is automatically recognized by IDA as a NOP and correctly shown:

```
MOV R0, R0 -> 00 00 A0 E1
```

The finally in order to skip the identified call it's enough to substitute it with a MOV R0, R0 or a MOV R1,R1.

```

loc_1004C1C8
ADD R0, R5, #4
ADD R1, R5, #0x150
BL FileExists__9Baf1UtilsRC3RFsRC7TDesC16 ; Baf1Utils::FileExists(RFs const &, TDesC16 const &)
CMP R0, #0
LDREQ R0, =0xFFFFEFC
BLEQ Leave__4User ; User::Leave(int)
MOV R0, R5
NOP
B loc_1004C1F8

```

This makes the program become fully registered, just because it doesn't call the protection manager. The only thing left is a warning message the first time the application is run, reporting that the application expired.

²³ Remember that the program is Little Endian, so reverse the value



6.2.2 Lesson learnt

Sometimes developers seem to forget that all the complex protections have an entry point or a single returning function. These bottle necks are useful to skip the complex protections simply not calling them. Also for this situation, before being threatened by the complexity of the protection framework the authors used, try analyzing how it is called/used by the application.

6.3. A Complex Protection with the classical heel of Achilles: ProfiMail 2.56

Profimail (<http://www.lonelycatgames.com>) is an excellent mail client for Symbian applications. It uses its own graphical routines to print text, create dialogs and so on. The result is a totally different aspect with respect to the other competitors, and on the other hand a lot of difficult code to analyze. There are not apparently useful calls to the system libraries and IDA poorly fails when analyzing it.

There is, since early versions, a keygen which generated the correct serial numbers starting from the phone's IMEI, but despite the generated serial codes are still valid for more recent versions, since version 2.5x they added an online check of the IMEI, which checks if the IMEI is registered into a remote database of "legal" IMEIs. This appears as dialog box reporting that the IMEI is not registered on the remote database. Then the program won't download mails.



Fortunately they did a big mistake in the protection scheme (for version 2.56) [37] which I will show here. They have now corrected it (but similar tricks are still possible).

These are the installed Files:

[Data]

- Alert.mid
- PM_S60.dta
- shop.txt

- ProfiMail.aif
- ProfiMail.app
- ProfiMail.rsc

The file PM_60.dta under the folder Data is indeed a zip file (it's enough to see its early bytes to understand it). Inside this file there are the graphics, fonts and local languages.

[demo]

- demo1.html
- demo2.html

```

demo3.html
lcg.jpg
more.html
On Phone.jpg
SM_Converter.jpg
SM_Player1.jpg
SM_Player2.jpg
UltraMP3_Screen.jpg
UltraMP3_Skin.jpg
[lang]
czech.txt
english.txt
finnish.txt
french.txt
german.txt
hungarian.txt
italian.txt
latvian.txt
polish.txt
portuguese.txt
russian.txt
slovak.txt
Font6x8_x.pcx
Font7x10_x.pcx
Font9x12_x.pcx
Font12x16_x.pcx
Font16x21_x.pcx
globe.pcx
globe_a.pcx
icons.pcx
icons_a.pcx
Logo.jpg
smallglobe.pcx
spr.txt
Symbols.pcx
Symbols_a.pcx

```

The interesting thing that one should immediately notice is that the Error MessageBox text, reported by the application, is always in English, no national settings change it. This is really important, because clearly shows that the message doesn't depend on the resources into the PM_60.dta file.

What I then thought was to modify the language files in order to de-align the resources of the program and see what happens. The main program loads the national resources from the .dta file and adds them to the internal resources, appending before or after the internal resources. De-aligning them is a tentative I did to understand if they were appended before or after.

What I did is to modify the english.txt file (it is a Unicode text file). It ends as following:

```

||-----|This is partially downloaded message.|Choose 'Download' from menu to
download entire message.|
Select provider:
Other
User manual
Auto-start
Start application automatically when device starts.
Copy marked to:
Move marked to:
Previous
Next

```

What I did is to eliminate the last line containing the "Next" string leaving the file with 387 lines instead of 388.

Once updated the .dta file and uploaded on the phone I discovered that Profimail was no more complaining with the Error messagobox. Excellent!

What happened? The internal resources are appended after the string read from the .dta file, then removing one line from the English.txt file resulted into a de-align of all the others. Apparently the string reporting the error was the last one of the list then now the API responsible of its creation just receives a NULL string and doesn't appear, because as a NULL argument in its call.

Following versions instead of changing the mechanism only added a local check on the number of lines a language file should have. When there are less than 388 lines the program complains with a system error which forces its immediate closure.

"Closed Appl. Profimail Bad Texts 0" ;388d = 184h

Of course there is a solution also for this problem, but I will not mention it here (the number of lines of a language file is stored somehow, somewhere..).

6.3.1 Packaging the new sis file

If you took a lot at the Profimail distribution sis, you should noted that the application installs something into the C disk and something into the installation disk you chosen. Then recreating the sis file requires a little more attention. We will again use the MakeSIS used in Section 3.1.3, but with different settings, see Figure 33.

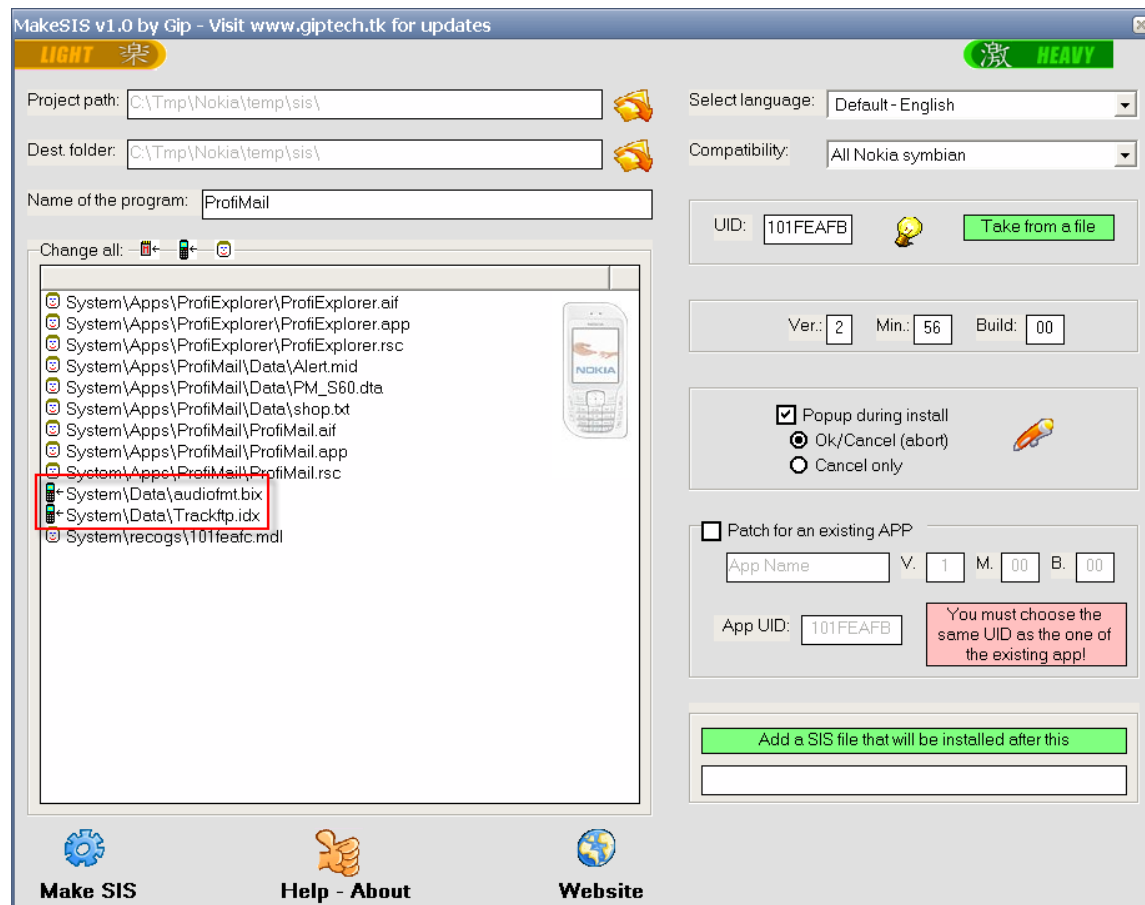


Figure 33 - Creating a new SIS for ProfiMail

To help you out further, the file structure I created on disk is shown in Figure 34.

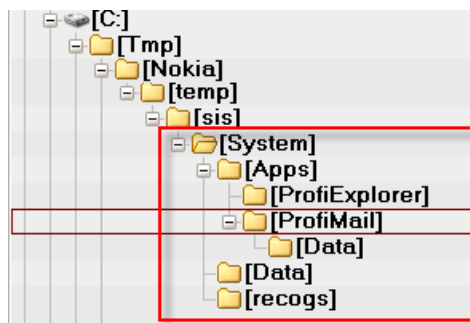


Figure 34 - File structure used to re-create Profimail sis file

6.3.2 Lesson learnt

Again a good example of BFR ("Bird Fly Reversing" ®). This application doesn't offer that much grips to OS calls from where one can start analyzing, but the surrounding environment is interesting. A good reversing habit for Symbian is always to look the surrounding files: all the information are stored into local files (there's no registry with Symbian). A fast search of what one can do managing the data on which the application works is always good. Creating SIS files also require attention on where the files must go, sometimes data are needed to go into a specific folder.

7. Conclusions & Further Readings

This tutorial covered an introduction to the Symbian reversing world and instruments, giving also some practical examples. Of course there are out several applications more complex than these, but using this document you should start learning and advancing.

As said from the beginning there are not too much tutorials around and most of them went lost when some sites went offline. I had to search and retrieve them using the WayBack Internet Machine or Alexa archives. This is a short list of documents I found and you can use, mirrored on our site.

- http://arteam.accessroot.com/tools/symbian/Change_ThumbsMode.html
- http://arteam.accessroot.com/tools/symbian/Cracking_SymbianWare_Stacker_v2.02.txt
- http://arteam.accessroot.com/tools/symbian/Cracking_www.Cellsoft.com_-_Cellpoker_v1.02.txt
- http://arteam.accessroot.com/tools/symbian/Cracking_RDCalc_220.html
- http://arteam.accessroot.com/tools/symbian/Making_KeygenPatch_for_WarFare_Incorporated_110.html
- http://arteam.accessroot.com/tools/symbian/SIS_File_FormatVersion_110_May-04.html
- <http://arteam.accessroot.com/tools/symbian/bh-eu-06-Niemela-up.pdf>
- http://arteam.accessroot.com/tools/symbian/deeper_ARM_Assembler_Keygenning_DataSafeS60.html

Also this document on malware for Symbian [29] is interesting.

Finally there's a list of tutorials from argv written for the Serbian ezine Phearless (also deroko is a member of):

- http://www.phearless.org/i2/Symbian_OS_%20Under_the_hood.txt
- http://www.phearless.org/i3/Symbian_Micro_Kernel.zip
- http://www.phearless.org/i4/Symbian_C++_Reference_Part%201.txt
- http://www.phearless.org/i4/Symbian_Polymorphic_MDL.tar.gz

The text is not clear for all of us, but thanks to deroko most of these papers have been translated purposely for this tutorial!

- http://arteam.accessroot.com/tools/symbian/Symbian_Micro_Kernel_Eng.rar
- http://arteam.accessroot.com/tools/symbian/Symbian_C++_Reference_Part_1_Eng.txt
- http://arteam.accessroot.com/tools/symbian/Symbian_Polymorphic_MDL_Eng.rar

This ARTeam CrackMe for Symbian [30] has not been written by me but from a friend I knew, purposely for this tutorial. The protection scheme is quite classical: trial time limit and limit on the times you can run it, then a simple registration number routine, but the program is based on modern programming logic, based on events and messages.

Try to patch or keygen it.

8. References

Due to the extremely vanishing nature of most of the links (some have been found by only thanks to Alexa - <http://www.alexacom.com> - and WayBack Machine Internet Archive - <http://www.archive.org/>) I mirrored on our web site some information and tools.

- [1] Alexander Thoukydides, SIS File Format, <http://homepage.ntlworld.com/thouky/software/psifs/sis.html>
- [2] Nokia Symbian SDK, http://seap.forum.nokia.com/main/resources/tools_and_sdks/listings/index.html
- [3] Nokia S60 Platform SDKs for Symbian OS, for C++,
<http://seap.forum.nokia.com/info/sw.nokia.com/id/4a7149a5-95a5-4726-913a-3c6f21eb65a5/S60-SDK-0616-3.0-mr.html>
- [4] SISView, <http://www.dalibor.cz/epoc/sisview.htm>
- [5] SYMBFS - Symbian Filesystem plugin for Total Commander, mirrored here
http://arteam.accessroot.com/tools/symbian/wfx_SymbFSPlgOnly04.zip
- [6] UnmakeSIS, originally here <http://mitglied.lycos.de/atzipzw> but now the site is gone, so take it here
<http://arteam.accessroot.com/tools/symbian/UnMakeSISv0.21.zip>. The tool is now commercial (greatly improved) and available here: <http://www.atz-soft.com>
- [7] uNsis by the3sky, www.3g365.com or
http://arteam.accessroot.com/tools/symbian/uNsis.en.v2.5_EN_by_the3sky.zip
- [8] MakeSis v1.0 by Gip, <http://www.giptech.tk> or
http://arteam.accessroot.com/tools/symbian/makesis_by_gip_last.zip
- [9] Jarno Niemelä, Symbian Malware What It Is And How To Handle it, F-Secure Corporation, mirrored here
<http://arteam.accessroot.com/tools/symbian/bh-eu-06-Niemela-up.pdf>
- [10] +Phantasm ERL Resource Decompiler, <http://arteam.accessroot.com/tools/symbian/erl.zip>
- [11] RSC Editor 1.0 by P_Jack, <http://www.symbian-freak.com/downloads/rsceditor.htm>
- [12] UIQEssentials_Getting_started,
http://developer.symbian.com/wiki/download/attachments/1138/UIQEssentials_Getting_started.pdf?version=1
- [13] Resource file format .rss,
http://icon.donga.ac.kr/%BD%C7%BF%F8%C0%DA%B7%E1%BD%C7/%B1%E8%B1%E2%C7%F6/!symbian%20project/Symbian%20SDK/DevLib/v6onedocs/Common/ToolsAndUtilities/DevTools-ref/Tool_Ref_RSS-file-format.guide.html
- [14] An application for Series 60, a step-by-step example, http://www.newlc.com/article.php3?id_article=402
- [15] MBM Wizard, <http://www.symbian-freak.com/downloads/mbmwizard.htm>
- [16] SymPorts IDA script by djnz, <http://arteam.accessroot.com/tools/symbian/SymPorts.rar>
- [17] Symbian 7.0 S60 IDA New IDS files,
http://arteam.accessroot.com/tools/symbian/Symbian_7.0_S60_IDA_IDS_files.rar
- [18] Symbian Oss 7.0 Developer Library,
http://www.symbian.com/Developer/techlib/v70sdocs/doc_source/index.html
- [19] Shub-Nigurath, WinCE Beginner Tutorial 1, <http://tutorials.accessroot.com>
- [20] Shub-Nigurath et al., Primer On Reversing .NET Applications, <http://tutorials.accessroot.com>
- [21] MaDMA_n_H3rCuL3s, DotNET Reverse Engineering Episode 1, <http://tutorials.accessroot.com>
- [22] "ARM Architecture Reference Manual", ARM DDI 0100E,
http://www.arm.com/documentation/Instruction_Set/index.html
- [23] "ARM v5TE Architecture Reference Manual, Issue E", <http://www.arm.com/miscPDFs/14128.pdf> (it's a crypted doc, you have to use for example APDFPRP to decrypt and being able to copy&paste text from it)
- [24] Funkysh, "Into my ARMs. Developing StrongARM/Linux shellcode",
http://isec.pl/papers/into_my_arms_dsls.pdf or
http://arteam.accessroot.com/tools/symbian/into_my_ARMs.txt
- [25] eEye Binary Diffing Suite, <http://research.eeye.com/html/tools/RT20060801-1.html>
- [26] Nokia SDK online, <http://www.symbian.com/developer/techlib/index.asp>
- [27] JAD, the Fasted Java Decompiler, <http://www.kpdus.com/jad.html>
- [28] Tim Lindholm, Frank Yellin, "The JavaTM Virtual Machine Specification, Second Edition",
<http://java.sun.com/docs/books/vmspec/2nd-edition/html/VMSpecTOC.doc.html>
- [29] Summer Brings Mosquito-born Malware, <http://www.informit.com/articles/article.asp?p=327990&rl=1>
- [30] ARTeam Crackme for Symbian 1.0,
http://arteam.accessroot.com/tools/symbian/CrackMe_Symbian_10.rar
- [31] Retrieving the device IMEI code, <http://newlc.com/Retrieving-the-device-IMEI-code.html>
- [32] How to Retrieve the IMSI Number, <http://newlc.com/How-to-retrieve-the-IMSI-number.html>
- [33] How to protect your application against a keygen, <http://newlc.com/How-to-protect-your-application.html>
- [34] How to identify a mobile at execution or installation time,
http://www.newlc.com/article.php3?id_article=161&var_recherche=machine+uid

- [35] Shub-Nigurath, UndlDC Program version 1.0,
http://arteam.accessroot.com/tools/symbian/undlDT_10_by_Shub-Nigurath.rar
- [36] Shub-Nigurath, UndlDC Program version 1.0 sources,
http://arteam.accessroot.com/tools/symbian/undlDT_10_src_by_Shub-Nigurath.rar
- [37] ProfiMail 2.56, http://arteam.accessroot.com/tools/symbian/profmail_symbian_lcg_2_56.zip
- [38] Application Resource tools guide,
http://www.symbian.com/developer/techlib/v70docs/SDL_v7.0/doc_source/ToolsAndUtilities/DevTools/index.html
- [39] SISXplorer, <http://www.symbian-toys.com/sisxplorer.aspx>
- [40] SISWare, <http://users1.nofoehost.com/CequenceTech/>
- [41] efd utility for IDA, http://hexblog.com/2005/11/how_to_unpack_xcpdat.html
- [42] Hex Workshop, <http://www.bpsoft.com/>
- [43] An introduction to Symbian C++ programming, http://www.supinfo-projects.com/fr/2005/introduction_symbian_en/
- [44] DisAsm, <http://www.iota.demon.co.uk/psion/disassembler/disassembler.html>
- [45] ARM Instruction Set Quick Reference Card, http://www.arm.com/pdfs/QRC0001_UALside2.pdf
- [46] Flander File Explorer, <http://arteam.accessroot.com/tools/symbian/FileExplorer.v1.0.rar>
- [47] VNavigator Siemens Obex File System, <http://www.totalcmd.net/plugring/vsofs.html>

9. Greetings

Greets goes out to the entire ARTeam family with its past, present and future members. A word of 'thank you' to the team members who took the time to read the alpha and beta versions of this document and also found some important errors (thanks flies to atzplzw and argv for the discussions on Symbian stuffs). If you've got suggestions and/or comments about this tutorial or want to contribute with something new or simply say hi, stop by at the forum and participate to the discussions there.

Document History

- **Version 1.0** First Public Release
- **Version 1.1** – error and fixes
 - ✓ Added IDA tutorials (§2.5.1) and IDA settings screenshots (§2.5.8)
 - ✓ Added 3D Minigolf (§6.2)
 - ✓ Added sections §6.1.2, 6.2.2 and 6.3.1 "Lesson Learnt", to each subchapter of §6.
 - ✓ Changed the chapter "Strings analysis of Symbian programs with IDA" §2.5.5 because I did a mistake on how IDA recognize strings.
- **Version 1.2** – more beginners things
 - ✓ Added details on the PC Suite for Nokia 6600 (§2.1)
 - ✓ Added references to SISXplore for paragraph "A Note on SISX Files" (§2.2.3)
 - ✓ Added how to modify Symbian applications using an Hex Editor (§3.1.2)
 - ✓ Added how to create multi-disk sis installations (§6.3.1)
 - ✓ Small corrections here and there
- **Version 1.3** – minor enhancements
 - ✓ Added details on the developing process (§1.1.2)
 - ✓ Added details on SymbFS and Obex (§2.2.1)
 - ✓ Updated on device tools adding DisAsm reference (§2.4)
 - ✓ Updated ARM tutorials references, adding quick references guides (§2.6)
 - ✓ Added more details on the steps required to find where imports are (§6.1)
- **Version 1.4** – added desquirr and example on its usage
 - ✓ Added introduction to desquirr (§ 2.5.6)
 - ✓ Used desquirr to help the reversing process (§ 3.1.1)

