FROM 0 TO 0DAY ON SYMBIAN

FINDING LOW LEVEL VULNERABILITIES ON SYMBIAN SMARTPHONES

Bernhard Müller

SEC Consult Vulnerability Lab, Vienna, 06/2009

V1.05 (public)

Table of Contents

ABSTRACT

Being the most widespread smartphone operating system[1], Symbian OS is a worthwile target for remote attacks. However, the obscurity of the operating system, combined with restrictions placed on end user devices and a lack of tools, make it very difficult for security researchers to work with Symbian based phones.

The goal of this whitepaper is to show that classic vulnerability analysis and exploitation is possible on Symbian OS smartphones. To this end, a set of methods and tools has been developed, and readily available standard software provided by Symbian has been modified to support debugging of memory mapped execute-in-place ROM. In this paper we will:

1. Show how to statically analyze XIP ROM images (dumping, restoring import- and export tables, searching for unsafe function calls)
2. Show how to enable run mode debugging of system binaries running from ROM with IDA Pro, by patching the AppTRK debug agent
3. Show other uses of the modified AppTRK. As an example, we will show a fully automated multimedia file fuzzer
4. List and analyze the results of fuzzing the video- and audio codecs shipped with current Nokia smartphones
5. Discuss further ideas and concepts, such as jailbreak shellcode, and an IRC bot trojan for Symbian

The paper aims to show that it is possible to find and exploit bugs on Symbian phones (even in preinstalled system applications) without having access to special development hardware, and that exploits and worms similar to those found on desktop systems may be possible on Symbian based smartphones.

INTRODUCTION

THE STATE OF VULNERABILITY RESEARCH ON SYMBIAN

Symbian OS is an unfriendly place for the security researcher. While we often see system processes crash on Symbian smartphones[2], a huge effort is required to figure out what is actually going on when it happens. One big reason for this is that hardly any useful tools are available to researchers. Many of those that are available are hard to use or don't work properly. In addition, the tools provided by Symbian, as well as the smartphones themselves, contain security mechanisms that prevent tampering with system applications.

[1] 52% share of the smartphone market as of April 2009, source: http://metrics.admob.com/wp-content/uploads/2009/05/admob-mobile-metrics-april-09.pdf

[2] Actually, this happens unintentionally all the time! On the first day I got my N96, I already managed to crash its Bluetooth stack, by attempting to sync my contacts via SyncML. So, in a way, I found the first "exploit" within two hours of working with the N96 (I found out later that this specific crash is caused by a user panic 23 exception, so this one is not really exploitable).

There also exists a myth that traditional low level vulnerabilities, such as buffer overflows, do not exist on Symbian OS smartphones[3]. Besides the steep learning curve required to work with Symbian OS and the lack of tools, this may be an additional factor that keeps security researchers off the operating system.

As a result, only a few low level vulnerabilities on Symbian OS smartphones have been discovered. In fact, a vulnerability search for "Symbian" on SecurityFocus returns only four results, two of which describe potential memory corruption errors in preinstalled system components[4]. Both vulnerabilities have been termed "denial of service". It is however likely that they have been given this status because no research has been done into the root cause of the vulnerabilities. This theory is supported by the original advisory on the well known "Curse for Silence" exploit, in which the discussion of the bug is limited to the observable behavior of the phone[5]. It seems that the underlying error condition, possibly an exploitable buffer overflow, has not been investigated.

Some pioneer work, mainly regarding shellcode for Symbian, has recently been published by Collin Mulliner (1). Other known efforts concentrate mainly on hacking platform security, which is regularly done by an active Symbian modding scene.

THE TEST DEVICE

The test subject used in this project was Nokia's current generation multimedia smartphone, the N96. The phone comes with S60 3rd Edition Feature Pack 2, which is based on Symbian OS v9.3. The installed firmware version was 11.018.280.2 dating from October 2008. The following list contains the complete version information:

- Software version: 11.018
- Software version date: 13-09-08
- Custom version: 11.018.280.2
- Custom version date: 16-10-08
- Language Set 001
- Model Nokia N96
- Type: RM-247

Like other smartphones the N96 ships with a basic set of applications, like web browser, calendar, music- and media player, plus some additional multimedia apps, for example a mobile TV application that uses the built in DVB-H receiver. What makes this phone great for research purposes is its big 16 GB built in flash memory – we had to put lots of software and files on the

[3] While it is true that Symbian C++ idioms such as string descriptors offer protection from some forms of buffer overflows, other forms of memory corruption may occur. In addition, as will be shown in this paper, smartphones may ship with poorly ported software that uses unsafe standard C library functions.

[4] „Symbian S60 Malformed SMS/MMS Remote Denial Of Service Vulnerability", http://www.securityfocus.com/bid/33072

"Nokia Series 60 BlueTooth NickName Remote Denial Of Service Vulnerability", http://www.securityfocus.com/bid/12743

[5] http://berlin.ccc.de/~tobias/cos/s60-curse-of-silence-advisory.txt

phone, and this way we simply didn't run out of space. Other than that, the base OS is no different to the versions used on other smartphones.

ARMV5 VS ARMV6 ARCHITECTURE

It is worth noting here that, while all Symbian OS phones run on ARM based ASICs, it makes a big difference if the CPU built into the board is based on ARMv5 and lower, or ARMv6 and higher architecture. The MMU has undergone a radical change in the ARMv6 architecture, and Symbian provides a completely different virtual memory implementation for the new architecture.

In short, for ARMv6 based architectures the "multiple memory model" is used instead of the "moving memory model". The multiple memory model supports multiple page tables, whereas the moving memory model moves blocks of memory in and out of a global page table, and a physically tagged cache is used instead of a virtually tagged one. This would not bother us as far as this paper is concerned, except that the memory mappings are also completely different in the multiple model. Because the Nokia N96 runs on ARMv5 based hardware, all virtual memory addresses mentioned in this paper apply to the moving memory model.

Another new feature in ARMv6 is that the page table permissions have been enhanced with a never execute bit. This is no problem in the scope of this paper, since we do not attempt to execute code from data pages. But it is a problem that has to be circumvented in ARMv6 architecture if we want to write an actual exploit.

However, in principle everything discussed in this paper also applies to smartphones running on ARMv6 based ASICs.

STATIC ANALYSIS OF XIP ROM IMAGES

On Symbian devices, preinstalled applications are stored on NOR flash in the form of prelinked execute-in-place (XIP) images. XIP refers to the ability to be executed directly out of the ROM memory. The XIP ROM image format is based on the standard EPOC E32Image executable, but there are some differences that affect the static analysis process.

During runtime, the ROM code is mapped directly to a fixed address within the global directory of the virtual memory map. For this to work, the executable images must already be relocated. Unfortunately for us, this means that nearly all additional information is discarded at build time.

The EPOC ROM-building tools perform the relocation and strip the images of their relocation information. The import and relocation sections of an executable are no longer needed when the executable is bound into a ROM. These sections are removed as part of the ROM building process. Additionally, EPOC uses link-by-ordinal exclusively, which means that no function names are contained in the export directories of system DLLs.

In this chapter, we will show how to restore most of the lost information for a static analysis of the ROM images. It will then be shown how to use the available information to search for potentially vulnerable components on our test device. Our goal is to find components that are most likely exploitable, so we have a target for the vulnerability analysis and debugging done in the later chapters.

An extensive primer on reverse engineering standard EPOC E32Image binaries has been written by Shub Nigurrath of ARTeam (2).

DUMPING THE SMARTPHONE ROM

As we already mentioned, the contents of the NOR flash ROM is mapped into the global directory, which means that it is seen by all processes running on the system. This is necessary because all processes need to be able to run code inside system DLLs, such as euser.dll, which are contained in the ROM. So, while we cannot directly access the protected files within the directory Z:\sys\bin\, we can dump the virtual memory region containing the ROM mapping.

The easiest way to do this is to simply dump the contents of the relevant memory region[6] into a file, and then extract the files contained in the ROMFS image. Tools that do this are already available, e.g. DumpTools by Zorn[7].

In the firmware dump of the Nokia N96 we found a total of 3.287 DLL and EXE files – that's more than in the Windows XP System32 directory! We can safely assume that at least some of them must be broken.

[6] The starting address of the ROM mapping can be obtained with the UserSvr::RomHeaderAddress() API. In S60 3rd Ed. FP2, the ROM is mapped at address 0xF8000000.

[7] I could not find an "official" link for DumpTools, but it can easily be found via Google or a search on http://www.symbian-freak.com

Now let's have a look at the disassembly of a typical EPOC ROM image in IDA Pro. In this example we will disassemble the Bluetooth engine server (btengsrv.exe). We'll disassemble a random piece of code inside this executable at address FA715BC4.

```
.text:FA715BC4 sub_FA715BC4                            ; CODE XREF: sub_FA715BF4+3A↓p
.text:FA715BC4                 PUSH    {R4-R6,LR}
.text:FA715BC6                 LSLS    R4, R1, #0
.text:FA715BC8                 LDR     R0, =dword_FA718564
.text:FA715BCA                 LSLS    R5, R2, #0
.text:FA715BCC                 LDR     R0, [R0,#0x14]
.text:FA715BCE                 BLX     sub_FA7180B0
.text:FA715BD2                 LSLS    R6, R0, #0
.text:FA715BD4                 LSLS    R2, R4, #0
.text:FA715BD6                 MOVS    R1, #5
.text:FA715BD8                 BLX     sub_FA7180C8
.text:FA715BDC                 BLX     sub_FA717E08
.text:FA715BE0                 LSLS    R2, R5, #0
.text:FA715BE2                 MOVS    R1, #6
.text:FA715BE4                 LSLS    R0, R6, #0
.text:FA715BE6                 BLX     sub_FA7180C8
.text:FA715BEA                 BLX     sub_FA717E08
.text:FA715BEE                 BLX     sub_FA717D18
.text:FA715BF2                 POP     {R4-R6,PC}
.text:FA715BF2 ; End of function sub_FA715BC4
```

Figure 1: Disassembly of a function in the btengsrv.exe XIP ROM image

As can be seen in the disassembly there are some function calls, but we can't really tell if those are library functions since IDA Pro cannot resolve any import names. We will soon see why. When we try to follow these function calls, we will end up at an import jump table starting at address FA717B50:

```
.text:FA717B50 sub_FA717B50
.text:FA717B50                 LDR     PC, =0xF855E9B5
.text:FA717B54 dword_FA717B54  DCD 0xF855E9B5
.text:FA717B58 sub_FA717B58
.text:FA717B58                 LDR     PC, =0xF8550561
..text:FA717B5C dword_FA717B5C  DCD 0xF8550561
.text:FA717B60 sub_FA717B60
.text:FA717B60                 LDR     PC, =0xF855061F
.text:FA717B64 dword_FA717B64  DCD 0xF855061F
(…)
```

Listing 1: XIP Rom disassembly without annotation

Since the image is already relocated, the jump table contains stubs that jump directly into other locations within the ROM mapping. Unfortunately, it is impossible to apply any generic signatures here, because the ROM images may be compiled differently for a given smartphone firmware version, i.e. the jump addresses will vary from phone to phone.

To solve this problem and make the disassembly more readable, we have to somehow collect the necessary symbol information specifically for our ROM build, and make it available to IDA. We implemented this by writing a script that collected all the available addresses for our specific firmware, added symbol names from the S60 SDK if available, and stored everything into a

SQLite database, which could then be accessed by IDA Pro, but which also proved useful for other purposes[8]. The basic functionality of the script will be described in the next section.

BUILDING A DATABASE OF ROM IMAGE SYMBOLS

In order to be able to map function addresses to function names, ordinals and DLL names, we will need the following information:

- Function addresses and ordinals of all exports in DLLs of our ROM dump: Can be extracted from the ROM images themselves.
- Function names for system libraries: For many important DLLs, such as euser.dll, the ordinal-to-name mappings can be found inside the import libraries found in public SDKs. For S60, these SDKs can be obtained at the Forum Nokia website[9].

To be able to extract the list of exports from the ROM images we have to look inside the image headers. Symbian ROM images have a TRomImageHeader, which is basically a shorter version of the E32ImageHeader used in normal EPOC E32 images. Its structure is shown in Figure 2.

Offset (hex)	Description of field
00	UID 1
04	UID 2
08	UID 3
0C	Checksum of UIDs.
10	Entry point of this executable (absolute address).
14	This executable's code address.
18	This executable's data address.
1C	Code size (includes constant data).
20	Text size (code size – size of constant data).
24	Data size.
28	BSS (zero-filled data) size.
2C	Heap minimum size (only needed for EXEs).
30	Heap maximum size (only needed for EXEs).
34	Stack size (only needed for EXEs).
38	Address of DLL reference table. (This is a list of the DLLs referenced by this executable which have static data.)
3C	Number of functions exported by this executable.
40	Export directory address.
44	Security information (capabilities, secure ID, vendor ID).
54	Version number of the tools used to generate this image file.
58	Flags field (see below).
5C	Priority of this process (only needed for EXEs).
60	Data and BSS linear base address – where this image file expects its data to be when it runs.
64	Next extension. Address of ROM entry header of subsequent extension files. This field is only used if there is more than one extension. The first extension is found using the `TRomHeader`.
68	A number denoting the hardware variant – used to determine if this executable can run on any particular system.
6C	The total data size (including space reserved for DLLs – for fixed address EXEs in moving memory model).
70	Version number of this executable – a 16-bit major and a 16-bit minor version number. This is used in link resolution.
74	Address of exception descriptor for this image (used in C++ exception unwinding). Zero if no exception descriptor present.

Figure 2: TRomImageHeader. Source: Symbian OS Internals (3)

The number of exports and the address of the export directory can be found at offset 0x3C into the file. The Perl code shown in listing 2 can be used to extract the addresses of all entries in the export table.

[8] In this chapter, we show two applications of the symbols database. It has however been useful in a number of other ways in this project, for example looking up function and DLL names when inspecting crash dumps.

[9] http://www.forum.nokia.com/Tools_Docs_and_Code/Tools/Platforms/S60_Platform_SDKs/

```
my $entryaddr = unpack("L", substr($img, 0x10, 4));

my ($nexports, $ expaddr) = unpack("LL", substr($img, 0x3C, 8));

print sprintf ("entrypoint address: 0x%04x\nnumber of exports: 0x%04x\nexport
directory: 0x%04x\n", $entryaddr, $nexports, $expaddr);

my $offs = $expaddr - $entryaddr + 0x78;

my @exports;

for (my $i = 0; $i < $nexports; $i++) {

    my $addr = unpack("L", substr($img, $offs + $i * 4, 4));

    push @exports, $addr;

    print sprintf ("0x%08x\n", $addr - 1);

}
```

Listing 2: Extracting the addresses of exported functions from a ROM DLL (Perl)

For each entry that is found, we try to look up the corresponding function names in the import libraries shipped with the S60 SDK. In the version used for this project, 3rd edition FP2, these libraries are in the directory:

C:\S60\devices\S60_3rd_FP2_SDK_v1.1\epoc32\release\armv5\lib\

A quick way to extract the export names from each import library is to use GNU nm, which can easily be installed via the Cygwin setup. The following command retrieves all symbol names for euser.dll from the corresponding import library euser.lib:

```
$ nm --demangle
/cygdrive/c/S60/devices/S60_3rd_FP2_SDK_v1.1/epoc32/release/armv5/lib/euser.lib

euser{000a0000}-1.o:
         U #<DLL>euser{000a0000}[100039e5].dll#<\DLL>1
00000000 t $a
00000004 t $d
00000000 T PanicTFixedArray()
00000004 t theImportedSymbol

euser{000a0000}-10.o:
         U #<DLL>euser{000a0000}[100039e5].dll#<\DLL>a
00000000 t $a
00000004 t $d
00000000 T CHeartbeat::NewL(int)
00000004 t theImportedSymbol

euser{000a0000}-100.o:
         U #<DLL>euser{000a0000}[100039e5].dll#<\DLL>64
00000000 t $a
00000004 t $d
00000000 T TMonthName::TMonthName(TMonth)
00000004 t theImportedSymbol
(…)
```

Listing 3: Extracting symbols from an import library

The S60 SDK provides import libraries for the most important components (including all the documented APIs), but it is far from complete[10]. For the remaining exports we automatically create short function names, in the form of [address]@[dllname]_[ordinal].

After gathering the export addresses and symbol names for each DLL found in our ROM dump, we store one entry per export into our symbols database, consisting of:

- Module name
- Function address
- Export ordinal
- Full function name
- Short function name

When the database has been built, we can use it to look up the list of export addresses for a DLL from our ROM, and their corresponding symbol names.

```
$ sqlite3.exe all.db
SQLite version 3.6.14.2
Enter ".help" for instructions
Enter SQL statements terminated with a ";"
sqlite> select dllname,addr,ordinal,fname from symbols where dllname='EUser.dll'
order by ordinal;
EUser.dll|4166372448|1|PanicTFixedArray()
EUser.dll|4166331014|2|CCirBuffer::Get()
EUser.dll|4166331050|3|CCirBuffer::Put(int)
EUser.dll|4166330976|4|CCirBuffer::CCirBuffer()
EUser.dll|4166330976|5|CCirBuffer::CCirBuffer()
EUser.dll|4166330998|6|CCirBuffer::~CCirBuffer()
EUser.dll|4166330990|7|CCirBuffer::~CCirBuffer()
EUser.dll|4166330990|8|CCirBuffer::~CCirBuffer()
EUser.dll|4166337630|9|CHeartbeat::New(int)
EUser.dll|4166337736|10|CHeartbeat::NewL(int)
EUser.dll|4166337772|11|CHeartbeat::RunL()
EUser.dll|4166337766|12|CHeartbeat::Start(TTimerLockSpec, MBeating*)
EUser.dll|4166337618|13|CHeartbeat::CHeartbeat(int)
EUser.dll|4166337618|14|CHeartbeat::CHeartbeat(int)
EUser.dll|4166337750|15|CHeartbeat::~CHeartbeat()
EUser.dll|4166337748|16|CHeartbeat::~CHeartbeat()
EUser.dll|4166337748|17|CHeartbeat::~CHeartbeat()
(…)
```

Listing 4 : Accessing the symbols database

With the help of the database, we should now be able restore the import jump tables and export tables in our disassembly.

ANNOTATING IMPORT AND EXPORT TABLES

IDA Pro provides several scripting interfaces. In our case, IDA Python is the most suitable, because it allows us to use the Python SQLite module to access our database.

The following small IDA Python script walks through the import jump table and uses the function addresses to do a name lookup in the database.

[10] It may be possible to get some of the missing symbol names from other SDKs (e.g. UIQ).


```
import sqlite3

conn = sqlite3.connect('c:\\n96_11_018.db')
conn.text_factory = str

c = conn.cursor()

ea = ScreenEA()

while (Word(ea) == 0xf004):

  addr = Dword(ea + 4) - 1

  c.execute('select shortname,fname from symbols where addr =?',(addr,))
  row = c.fetchone()

  print addr, row

  MakeNameEx(ea, row[0], SN_NOCHECK)
  MakeComm(ea,row[1]);

  ea += 8
```

Listing 5: Annotating the import jump table with IDA Python

If we take another look at the function at address FA715BC4 after running the script, we can see that most of the function names have been successfully resolved. It's now much easier to deduce what the function is actually doing (in this case, calling the central repository APIs).

```
.text:FA715BC4 sub_FA715BC4                            ; CODE XREF: sub_FA715BF4+3A↓p
.text:FA715BC4                 PUSH    {R4-R6,LR}
.text:FA715BC6                 LSLS    R4, R1, #0
.text:FA715BC8                 LDR     R0, =dword_FA718564
.text:FA715BCA                 LSLS    R5, R2, #0
.text:FA715BCC                 LDR     R0, [R0,#0x14]
.text:FA715BCE                 BLX     __CRepository__NewLC_centralrepository_16
.text:FA715BD2                 LSLS    R6, R0, #0
.text:FA715BD4                 LSLS    R2, R4, #0
.text:FA715BD6                 MOVS    R1, #5
.text:FA715BD8                 BLX     __CRepository__Get_centralrepository_9
.text:FA715BDC                 BLX     __User__LeaveIfError_EUser_593
.text:FA715BE0                 LSLS    R2, R5, #0
.text:FA715BE2                 MOVS    R1, #6
.text:FA715BE4                 LSLS    R0, R6, #0
.text:FA715BE6                 BLX     __CRepository__Get_centralrepository_9
.text:FA715BEA                 BLX     __User__LeaveIfError_EUser_593
.text:FA715BEE                 BLX     __CleanupStack__PopAndDestroy_EUser_203
.text:FA715BF2                 POP     {R4-R6,PC}
.text:FA715BF2 ; End of function sub_FA715BC4
```

Figure 3: Annotated API functions in IDA Pro

To select a target for the remainder of the paper, we need some way to determine which components are potentially vulnerable. We will simply use the symbols database to look for components that use insecure APIs, such as the string functions from the good old fashioned str(...) family[11].

In Symbian, there are two implementations of the standard C APIs. The old one, which is still shipped with devices running S60 3rd edition FP2 (at least it was available on our N96), is called estlib. Additionally, FP2 contains a more complete implementation of the POSIX standard, called P.I.P.S. ("PIPS is POSIX on Symbian") or OpenC. Both of them provide basic POSIX string functions, such as strcpy() and strcat().

To automate this, we first look up the addresses of the functions we are interested in, both inside estlib.dll and libc.dll. Then we parse the ROM images themselves for branch instructions to these addresses. This functionality can easily be built into another Perl script.

Let's now look at some of the results:

```
ldr pc, 0xf88a633d (strcpy) in mdfh264payloadformat.dll!
ldr pc, 0xf88aa203 (sprintf) in mdfh264payloadformat.dll!
ldr pc, 0xf88a62bb (strcat) in mdfh264payloadformat.dll!
ldr pc, 0xf88aa203 (sprintf) in xmlsec.dll!
ldr pc, 0xf88a62bb (strcat) in progdownfs.dll!
ldr pc, 0xf88a633d (strcpy) in progdownfs.dll!
ldr pc, 0xf88a633d (strcpy) in WebCore.dll!
ldr pc, 0xf88aa203 (sprintf) in mdfvidrender.dll!
ldr pc, 0xf88a633d (strcpy) in mdfvidrender.dll!
ldr pc, 0xf88a62bb (strcat) in mdfvidrender.dll!
ldr pc, 0xf88a62bb (strcat) in wmarender.dll!
ldr pc, 0xf88aa203 (sprintf) in wmarender.dll!
ldr pc, 0xf88a633d (strcpy) in wmarender.dll!
ldr pc, 0xf88a633d (strcpy) in vidsite.dll!
ldr pc, 0xf88aa203 (sprintf) in vidsite.dll!
ldr pc, 0xf88a62bb (strcat) in vidsite.dll!
ldr pc, 0xf88a62bb (strcat) in mp4fformat.dll!
ldr pc, 0xf88a633d (strcpy) in mp4fformat.dll!
ldr pc, 0xf88aa203 (sprintf) in mp4fformat.dll!
(...)
```

Listing 5: Partial list of invocations of unsafe string functions

[11] Vulnerabilities may also exist in other components. However, as we did not want to make our lives unnecessarily hard, we searched for the low hanging fruit (which incidentally is what an actual attacker would do).

As can be seen in listing 5, potentially dangerous string functions are imported by a lot of libraries that seem to be some sort of parsers, or codecs, for video and audio file formats (WebCore.dll also sounds interesting in the list above).

We can be almost certain that these modules are ports of standard C implementations that were originally created for PCs, and do not use the Symbian specific string handling classes. There is a good chance that, since these unsafe functions are used at all, at least some of the components are of poor quality, and may have bugs that could turn out to be exploitable.

Another factor that makes these components interesting are the potential attack vectors that would exist once a vulnerability is found. In modern smartphones, video and audio files can be embedded into MMS messages. Since the MMS viewer uses the same codecs to play video and audio files, any vulnerability in one of these codecs would be remotely exploitable.

This is why, for the rest of this paper, we will concentrate on the multimedia codecs contained in our firmware. The techniques presented in the later chapters can however be also applied to other components!

DEBUGGING HIGHLY PRIVILEGED PROCESSES

Now that we have determined that the multimedia codecs are probably broken, why not give it a shot and try to exploit them? An easy way to do this would be to create broken multimedia files, feed them into MediaPlayer.exe, and see what happens.

The only problem is, we can't really see what happens. The problems start as soon as we try to attach the IDA Pro Symbian remote debugger to the MediaPlayer.exe process running on the smartphone: Not only does the IDA debugger not work with current TRKs, but we can't attach to the MediaPlayer-process even with the Carbide C++ debugger. Also, for the processes we manage to attach to, the debugger does not show any code in the ROM section! We will deal with these problems in the next chapter.

RUN MODE DEBUGGING OVERVIEW

Run mode debugging, which is very common in the embedded device industry, refers to the method of debugging an application directly on the target device. The debugger is run on the host application (PC), and sends commands to an on device debug agent over a common communication channel, such as Bluetooth, IR or USB. The debug agent can be used to run programs and attach to processes, set breakpoints, and read or set registers and memory. It also reports back any relevant events, such as exceptions, back to the host system. Figure 4 shows the architecture of the run mode debugger.

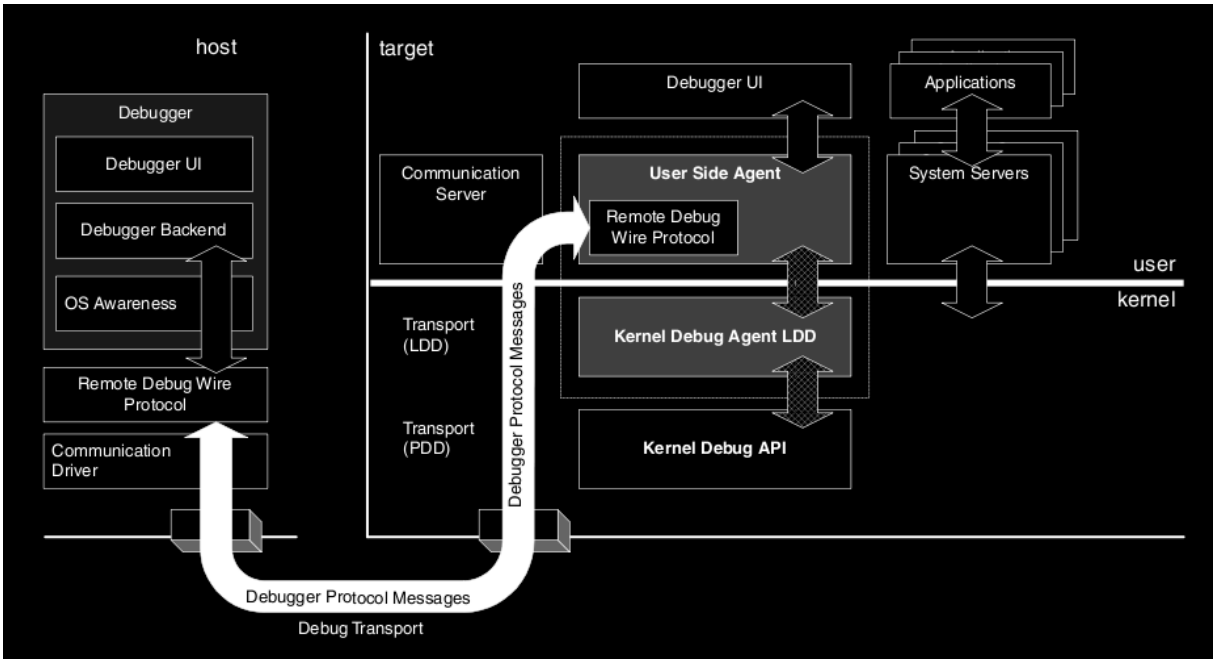

Figure 4: Run mode debugging architecture. Source: Symbian OS Internals (3)

In the case of Symbian, a run mode debugging facility is provided by the publicly available AppTRK. AppTRK is a target-resident debug agent that consists of a logical device driver (LDD), a DLL, and a GUI application that allows users to configure the application.

In principle, AppTRK remains transparent to the operating system and can be used to debug any application running on the device. However, it cannot be used to debug the kernel or device

drivers, since the debug agent itself requires kernel services to run. The AppTRK interface is partially documented in the CodeWarrior MetroTRK Reference (4).

In order to prevent evildoers from messing around with important system services, the AppTRK debug agent contains protection mechanisms that prevent the debugger from attaching to processes with specific capabilities. Additionally, it does not allow reading from the ROM memory region. Since all preinstalled applications and modules have their code in ROM, this means that only user installed applications can be debugged.

To overcome this problem, we will have to patch the debug agent kernel driver and remove these protections. But we cannot simply run a modified debug agent on the device – as it requires the TCB capability, it cannot be run without being signed with a Nokia phone manufacturer certificate. As the name implies, these certificates are only handed out to phone manufacturers. For this reason we have to apply a platform security hack before doing anything else.

HACKING THE PHONE

The following chapter starts with a short discussion of platform security. After that, we will describe how to exploit a known vulnerability to disable platform security on current S60 3rd edition FP2 smartphones[12].

PLATFORM SECURITY OVERVIEW

Platform security is the means by which Symbian OS attempts to protect its integrity. It has mainly been implemented to protect unsuspecting smartphone users of bad surprises, such as an exorbitant phone bill caused by malicious software running on the phone.

In Symbian, the process is the main unit of trust. Each process running on Symbian is assigned a set of capabilities, which are used to decide which system APIs the process can access. Some of these capabilities can be granted by the user, but others require the application to prove its level of trust with a valid signature. The most important APIs, those within the innermost layer of trust (the "trusted computing base") can only be accessed with approval from Symbian Corporation or the phone manufacturer.

[12] This worked on our Nokia N96 with its original S60 3rd Ed. FP2 firmware installed, which was the most current version at least until March 2009. I did not want to update the firmware during this project, and I'm not sure if the bug has been fixed in newer versions. But this is actually not a big concern, since platform security hacks are found on a regular basis! See http://www.symbian-freaks.com/ for current discussions on this topic.

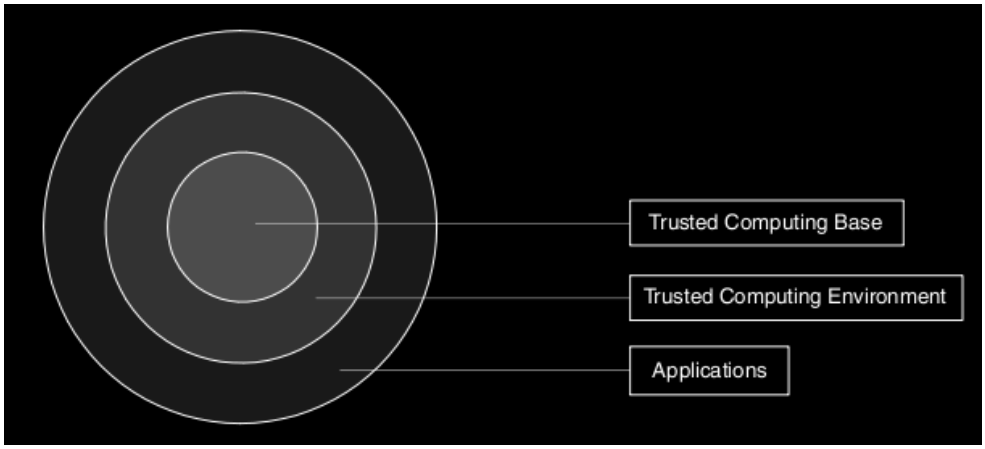

Figure 5: Layers of trust in platform security. Source: Symbian OS Internals (3)

Let's have a look at how this has been implemented. As we already said, each process has its own set of capabilities. The programmer has to decide at build time which capabilities will be required. The chosen capabilities are then hardcoded into the header of the executable image. When a program is started, and the image is loaded into memory, the loader associates the new process with the capabilities specified in the header. The capabilities are set only once and cannot be changed during runtime.

Capabilities are specified for executable files as well as for DLLs. To prevent malicious code from being loaded into privileged processes, a process is not allowed to load a DLL that has a smaller set of capabilities than itself, and a DLL cannot statically link to another DLL with a smaller set of capabilities.

So, what prevents us from simply giving our executable any capabilities we want? The answer is a feature called "data caging". The loader will refuse to load executables from directories other than the \sys\ path. This directory is protected by the fileserver[13] and can only be written to by processes with the TCB (Trusted Computing Base) capability. Installing executables and libraries into this directory is never done directly, but is done by the install server.

	Capability required to:	
	Read	**Write**
\resource	none	TCB
\sys	AllFiles	TCB
\private\<ownSID>	none	none
\private\<other>	AllFiles	AllFiles
\<other>	none	none

Figure 6: Data caging / capabilities overview. Source: Symbian OS Internals (3)

[13] In Symbian, client server IPC is used for accessing files and many other operating system services. For example, to open a file for reading, a process has to create a session with the fileserver. For a detailed description of the Symbien client/server IPC concept, see (3).

In order to install software on the device, users have to provide so called SIS packages to the install server. The SIS package contains the binary files as well as resources and metadata required for the installation, and can be signed with a software certificate. At the time of installation, the install server checks the set of capabilities requested by the binaries contained in the package. It then decides whether to allow or disallow the installation, based on the configuration of the device and on the signature of the SIS package.

Code signing certificates for development or publishing purposes can be obtained at the Symbian Signed website[14]. We won't go into detail about the different certificates here. It is however important to note that the most important capabilities, TCB, AllFiles and DRM, cannot be obtained through Symbian Signed without manufacturer approval. The table in Figure 7 shows an overview of the available signing options and capabilities.

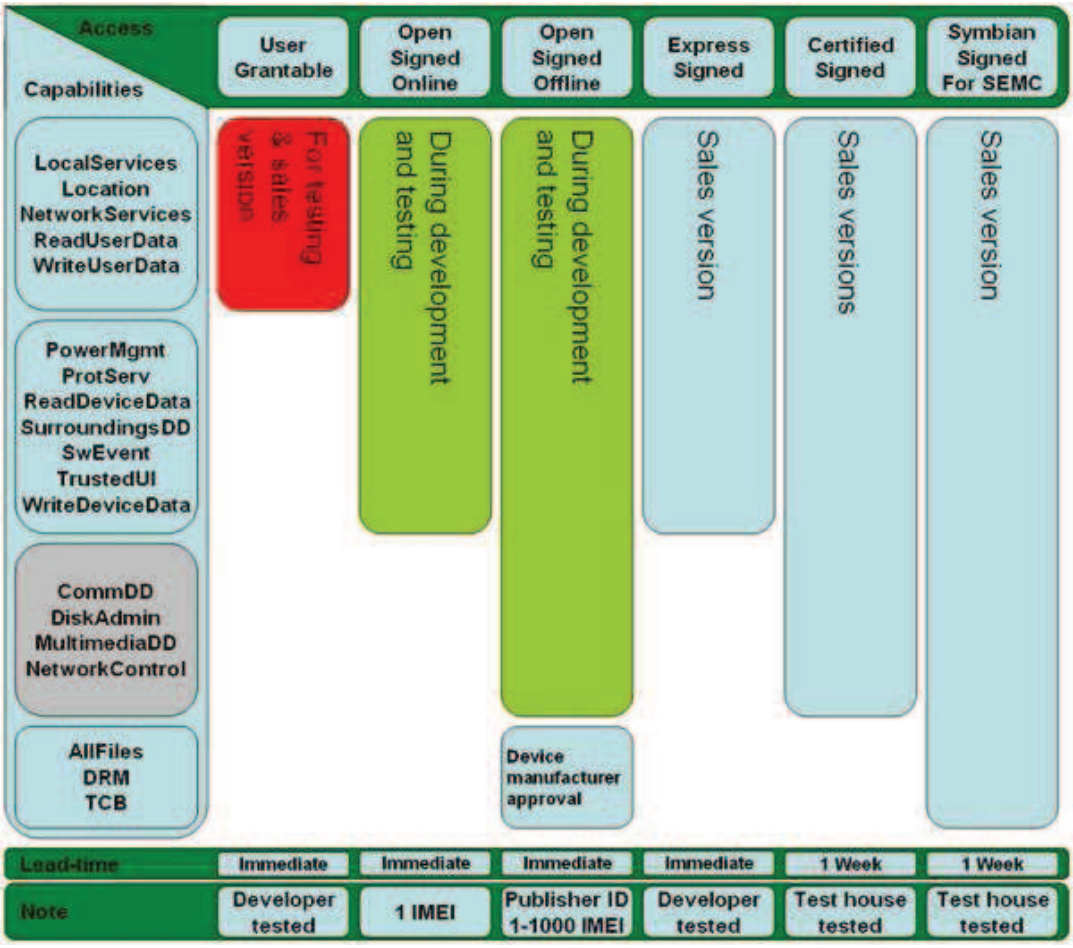

Figure 7: Symbian signed grid

[14] https://www.symbiansigned.com/

PROBLEMS WITH PLATFORM SECURITY

Observant readers may already have noticed that the concept of platform security contains some pretty 'unique' ideas, a characteristic that applies to most of Symbian OS[15]. The main problem with platform security is that it is much more complex than necessary. If trusted computing is really needed in Symbian OS, why not require the binaries themselves to be signed? This way, a single security check would take place in a single component (the loader), at the moment before the program is executed. Instead, the integrity of the system depends on a lot of components working together, and failure in any of these components compromises the whole system.

Additionally, the path based data caging mechanism does not seem like a very robust security strategy. It is based solely on the assumption that executable files cannot be placed into certain directories.

The first problem is that, obviously, this assumption does not hold true for removable media. The Symbian designers thought of this and implemented a hashing mechanism: Hashes of executables installed on removable media are calculated and stored in a database on the integrated flash memory, and the loader checks it for a valid hash before executing programs from removable media. But the fact that this secondary measure is even necessary shows that the idea of path based data caging itself is worthless! If we have to store hashes anyway, then we could also calculate a hash for all binaries that are allowed to run, store that in a single locked system file, and throw away the complicated path protection rules, with the same effect.

Complexities like these also make the system prone to logical errors. How do we decide, in the example above, which media is removable media and which is not? These complexities result in bugs similar to the one shown in the next chapter.

THE MAPDRIVES EXPLOIT

The platform security vulnerability discussed in this chapter was discovered by Alex_N70 and has been originally posted in the Italian Nokioteca forum in January 2009[16].

The MapDrives method exploits a flaw in the executable loader. The problem lies in the fact that a process with the DiskAdmin capability can access an API that maps subdirectories to unused drive letters. By creating the subdirectories \sys\bin\ in an existing directory, and mapping this directory to a new drive letter such as Y:, the attacker can place executable files into a path that, under certain circumstances, seems like a valid executable path to the loader.

It is however not possible to execute files directly from the newly mapped \sys\bin\ directory. The attack only works if the loader is instructed to load an executable, but no specific path is specified.

As an example, let's assume that we place a file called "test.exe" inside the directory E:\hack\sys\bin\. We then map the directory E:\hack\ to drive Y:. By now calling RProcess::Create("test.exe"), we instruct the loader to execute a file called test.exe. The loader will now search the \sys\bin\ directories of all available drives for a file of this name. At some

[15] One particularly funny idiom is the Symbian C++ exception handling with Leaves / CleanupStack. This and other features of Symbian C++ are well described in (6).

[16] http://www.nokioteca.net/home/forum/index.php?showtopic=143499

point it will find our executable, Y:\sys\bin\test.exe. In this case, the loader considers it a valid executable file, and loads it into memory, assigning it any capabilities we may have specified.

The following listing shows Symbian C++ code that establishes a session to the file server and subsequently uses the RFs::SetSubst() API to map a directory to all available drive letters.

```
LOCAL_C void MainL()
{
	_LIT(KPath,"E:\\subst\\");
	RFs fs;
	TInt i, err;
	CleanupClosePushL(fs);
	User::LeaveIfError(fs.Connect());
	for (i = 0; i < 25; i++) {
		err = fs.SetSubst(KPath, i);
	}
	fs.Close();
	CleanupStack::PopAndDestroy();
}
```

Listing 6: Mapping a subdirectory to drive letters

It is important to note that a process run this way can have any capabilities, including AllFiles, DRM and TCB.

Our main use of the exploit in this project was to break platform security on our test device – with platform security enabled, it would have been impossible to debug any of the more important processes. The techniques described in the following chapters can therefore be reproduced only on a hacked device. The way we did this, and also probably the easiest way at the time I'm writing this, is to use the MapDrives exploit explained in this chapter to install additional "Symbian A" root certificates on the device. Once these root certificates are installed, we can install SIS packages with any capabilities we want. You also do not have to do any coding yourself – readymade packages exist out there which perform the whole process automatically.

CRACKING APPTRK

Now that we can run modified software with any capabilities we want, we can try to create a modified version of the debug agent that we can later use to debug system processes. At first, let's look at the files contained in the AppTRK SIS package[17]. The most important ones are:

- trkdriver.ldd: TRK kernel driver that implements the debugging functionality
- trkengine.dll: This DLL exports the debugging API to user mode applications. It accesses the debug agent driver via the RBusLogicalChannel interface
- trkguiapp.exe: The executable that is used to initialize and configure the debugger

From a security perspective, it would make the most sense to put the protection mechanisms into the device driver, so let's assume that we can find them there.

[17] The version we used was AppTRK 3.0.8 for S60 3rd Ed. FP2.

REMOVING PROTECTION OF ROM MEMORY

The first thing we want to fix is the protection of ROM memory. One approach that can be used is to trace the function flow from where the TrkReadMemory message code is received by trkengine.dll, all the way until the memory is actually read by trkdriver.ldd, and see if there are any interfering checks in between.

From the metrotrk.h header file contained in the IDA SDK, we know that the TRK message code for TrkReadMemory is 0x10. There is a large jump table inside trkengine.dll that branches depending on this message code. When we start from there, we eventually end up at a call to the RBusLogicalChannel::DoControl() API that sends a request to the kernel driver. As expected, there are no checks in between as far as trkengine.dll is concerned.

However, if we follow the respective request handler inside trkdriver.ldd, we will find two functions that perform a suspicious series of register comparisons. Both functions take an integer argument and check if it is in a specific range, the first one representing the kernel data sections of the moving and multiple memory models, the second one the address ranges of the memory mapped ROM section. There is also a subsequent call to Kern::ThreadRawRead() that depends on the outcome of the comparisons, so we can safely assume that this is the check we are looking for.

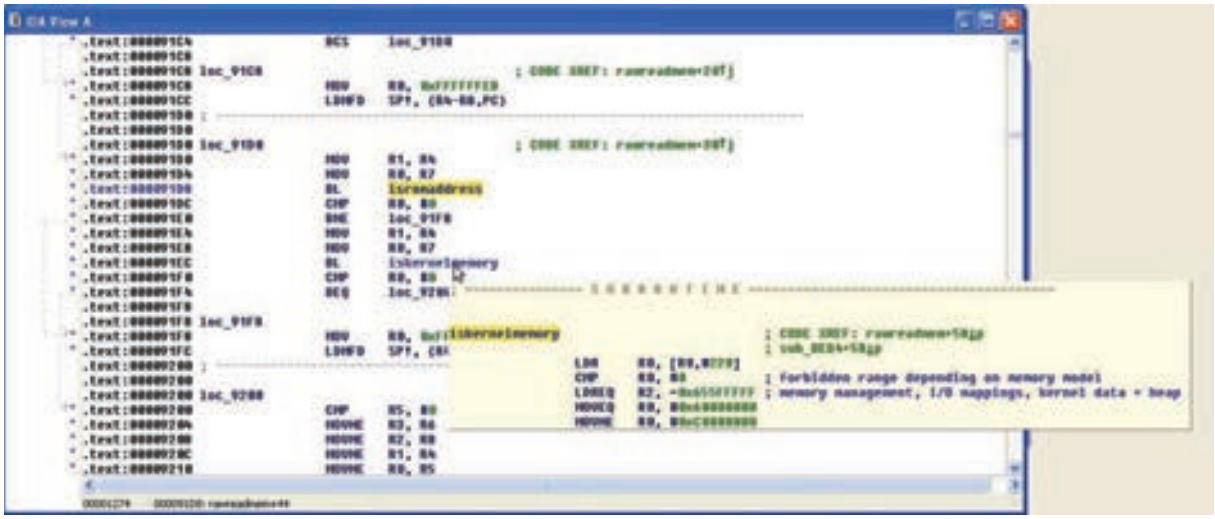

Figure 8: Branches to memory protection checks (already annotated)

We can disable this check by introducing a jump over both comparison functions. We place a branch instruction at file offset 0x1250 (virtual address 0x91B4):

```
11 00 00 ea (branch +44)
```

REMOVING PROTECTION OF SYSTEM PROCESSES

We can now read and debug within the ROM code range, but we still can't attach to MediaPlayer.exe, or any other process with the DRM, AllFiles or TCB capabilities!

This can also be fixed easily. The debug driver uses the DThread::DoHasCapability() API to check the capabilities of a process. There are only five uses of this API within the driver, three of which are in a single function at address 0xb714, and these are successive checks for the TCB, DRM and

AllFiles capabilities. The function is called only once at 0xb79c, so we simply replace this branch instruction with a NOP (we need a return value of 0).

The patch is applied at file offset 0x3838 (virtual address 0xB79C):

```
00 00 a0 e0 (mov r0,0)
```

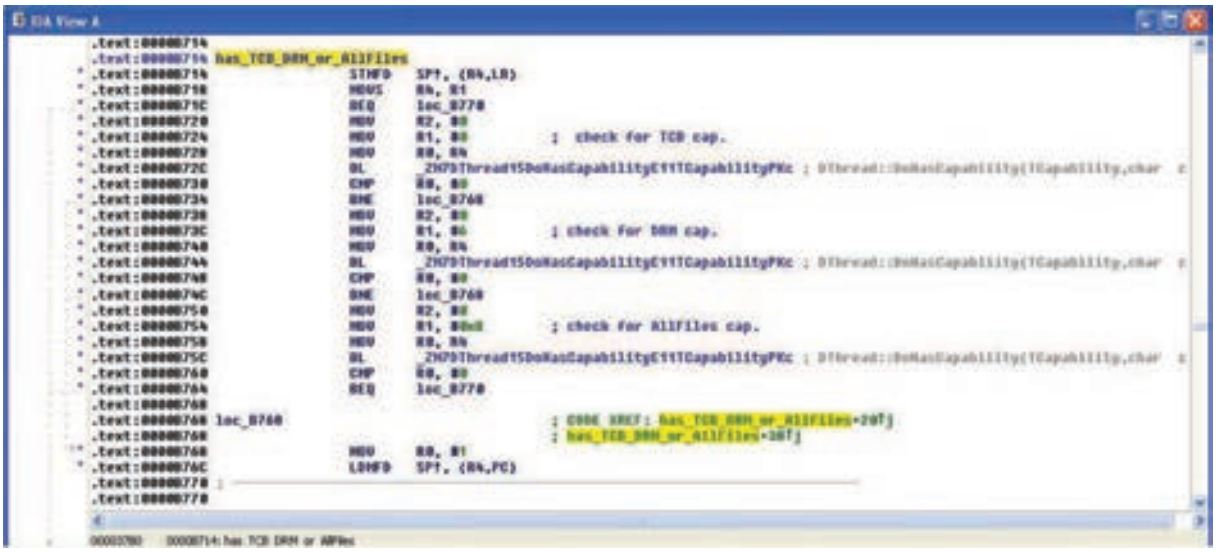

Figure 9: Disassembly of the target process capabilities check (already annotated)

TESTING THE PATCHES

To test the patches, we now replace the existing trkdriver.ldd in the C:\sys\bin\ directory on the smartphone[18].

With the patches in place, we can now try to load a ROM executable into IDA Pro, set some breakpoints in ROM, and attach to the corresponding process. As an example, we can try a system server, such as PhoneServer.exe. As can be seen in Figure 10, it works, including breakpoints and single stepping.

[18] A „special" file browser, such as Modo by Leftup, is needed for this. Why special? Remember the discussion of platform security? To write into the /sys/bin/ directory on any drive, a process needs the TCB capability. But such a process can not have a GUI, because none of the GUI DLLs has the TCB capability, and a process cannot load a DLL with lower privileges than itself! This is one of the odd effects of platform security. The Modo file browser solves this problem by being split into two components, a server and a GUI app.

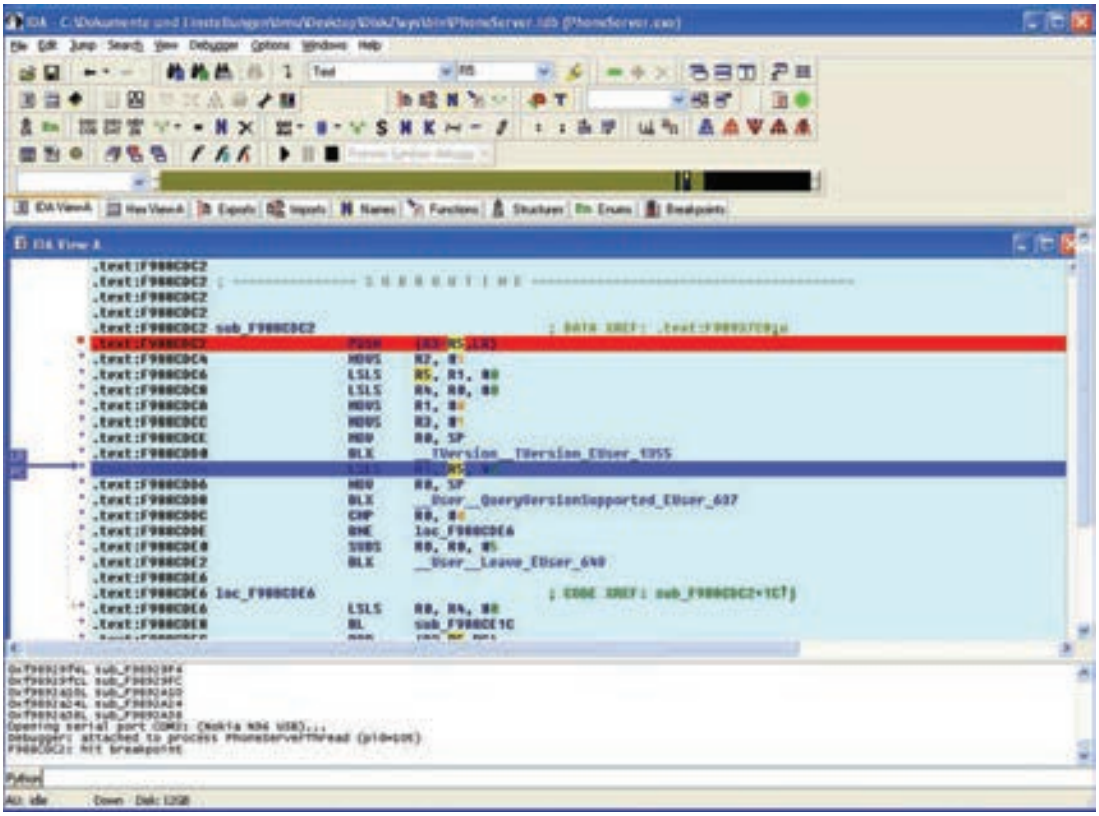

Figure 10: Single stepping within the ROM code of a system server, in IDA Pro

INTERACTING WITH APPTRK

So far so good, we have now opened up the system for debugging, so let's discuss some uses for AppTRK.

DEBUGGING WITH IDA PRO

The most obvious use of our cracked debug agent is remote debugging of processes with IDA Pro. This is useful to help reverse engineering the system, and also to investigate exceptions thrown by system processes or other applications running from ROM. For this purpose, current versions of IDA Pro contain a Symbian remote debugger plugin, which allows the user to conveniently debug processes from the familiar GUI.

One problem we ran into was an undocumented change of the AppTRK message protocol that seems to have been introduced in version 3. Sniffing the serial port communications between Carbide.C++ and AppTRK 3 showed that all packets contained an additional header that did not exist in prior versions. Since the header was missing from IDA's messages, the AppTRK did not respond at all.

Luckily, the complete source code for the Symbian debugger plugin is contained in the IDA Pro SDK, so we simply added some code to create the header and recompiled the plugin.

```
//--------------------------------------------------------------------------

// Add header

void metrotrk_t::prepend_hdr(void)

{

       size_t len = pkt.size();

       prepend_byte((uchar)len);

       prepend_byte((uchar)len >> 8);

       prepend_byte(0x90);

       prepend_byte(0x01);

}
```

Listing 7: Code to prepend the additional header to a MetroTRK packet

The fix has been included in current versions of IDA Pro (>5.5)[19].

Other than that, debugging should work normally for most processes, including system servers. It is however not possible to connect to processes that are part of the kernel, such as ekern.exe and efile.exe, since this will hang the entire operating system, and with it the debug agent.

CONTROLLING APPTRK

Besides GUI based debugging, there are plenty of other uses for AppTRK. For this reason, we built a C++ DLL that exports the metrotrk_t class, which provides an interface to AppTRK. Most of the required code already existed in IDA's MetroTRK implementation, which we were kindly allowed to borrow by the author, so all that was required was some restructuring and porting.

The DLL exports an interface that is mostly self-explanatory. To use it, an application has to instantiate an object of the metrotrk_t class, and provide a function pointer to the handle_notification() callback function. This function is called when a notification is received from AppTRK.

Amongst other things, the following can be automated by using our DLL:

- Copying files to and from the device
- Installing .SIS packages
- Launching and terminating applications
- Getting a list of running processes
- Setting and deleting breakpoints
- Stopping and resuming threads of an attached process
- Reading and writing registers and memory of an attached process

These functions should suffice for most dynamic vulnerability analysis tools we might want to create[20]. As an example, a fully automated file format fuzzer is shown in the next section.

[19] http://www.hex-rays.com/idapro/55/index.htm

The following listing shows the public interface of the metrotrk_t class.

```
struct  __declspec(dllexport) metrotrk_t
{
(…)
public:
  metrotrk_t(void) {fp_handle_notification = NULL; debug_debugger=0;}
  ~metrotrk_t(void) { term(); }
  void setdebug(bool which);
  bool init(int port);
  void term(void);
  bool reset(void);
  bool ping(void);
  bool connect(void);
  bool disconnect(void);
  bool set_notification_func(handle_notification_func h);
  bool support_mask(uchar mask[32], uchar *protocol_level);
  bool cpu_type(trk_cpuinfo_t *cpuinfo);
  int open_file(const char *name, trk_open_mode_t mode);
  ssize_t write_file(int h, const void *bytes, size_t size);
  ssize_t read_file(int h, void *bytes, size_t size);
  bool seek_file(int h, uint32 off, int seek_mode); // SEEK_...
  bool close_file(int h, int timestamp);
  bool install_file(const char *fname, char drive);
  bool create_process(
        const char *fname,
        const char *args,
        trk_process_info_t *pi);
  int attach_process(int pid); // returns tid
  bool resume_thread(int pid, int tid);
  bool step_thread(int pid, int tid, int32 start, int32 end, bool stepinto);
  bool suspend_thread(int pid, int tid);
  int add_bpt(int pid, int tid, int32 addr, size_t len, int count, bool
thumb_mode);
  bool del_bpt(int bid);
  bool change_bpt_thread(int bid, int tid);
  bool terminate_process(int pid);
  ssize_t read_memory(int pid, int tid, int32 addr, void *bytes, size_t size);
  ssize_t write_memory(int pid, int tid, int32 addr, const void *bytes, size_t
size);
  bool read_regs(int pid, int tid, int regnum, int nregs, uint32 *values);
  bool write_regs(int pid, int tid, int regnum, int nregs, const uint32 *values);
  bool get_process_list(proclist_t &proclist);
  bool get_thread_list(int pid, thread_list_t *threadlist);
  bool poll_for_event(int timeout);
  int32 current_pid(void) const { return tpi.pid; }
  bool recv_packet(uchar *seq, int timeout);
  bool send_reply_ok(uchar seq);
  uchar extract_byte(int &i);
  uint16 extract_int16(int &i);
  uint32 extract_int32(int &i);
  string extract_pstr(int &i);
  string extract_asciiz(int &i);
};
```

Listing 8: Public methods exported by the metrotrk_t class (C++ / WIN32)

[20] Unfortunately, although the message code exists, resetting the device is not supported by AppTRK. This is bad because some tasks, such as on-device fuzzing, tend to mess up the operating system, and in this case an occasional reboot would be beneficial. Maybe this feature can be introduced with further patches.

(AB-)USING APPTRK: AUTOMATED MULTIMEDIA CODEC FUZZING

In the previous chapters, we have used static analysis to search our ROM dump for suspicious uses of unsafe string functions, and determined that the multimedia codecs on our N96 don't look very trustworthy. We have also placed a modified debug agent on the device, and written a DLL that provides us with an interface. Now, we will put all the pieces together and use all this stuff to find some actual bugs. We should now easily be able to write a file format fuzzer that does all the work for us.

As the target process, we choose MediaPlayer.exe (RealPlayer), since this application uses the potentially vulnerable codecs to parse and playback multimedia files. It is important to note that MediaPlayer.exe is not the only application using them: The MMS viewer has the capability to play video files, and it loads the same DLLs for that purpose. Fuzzing MediaPlayer.exe is however easier to automate.

AUTOMATING THE FUZZING PROCESS

We will now use the class we created in the previous chapter to automate the fuzzing process. Since we want to create malicious multimedia files, we need some kind of file format fuzzer.

The goals for our fuzzer are the following: It should create the input files, upload them to the phone from the host PC, and have them loaded by MediaPlayer.exe. It should then determine if MediaPlayer.exe has crashed. If it has, it should determine the current register values and log the incident, and after that, continue the fuzzing process. It should also be able to recover automatically if the TRK crashes or the phone reboots, something that happens often in Symbian when it is put under stress.

Using the class we created in the previous chapter, we can easily start the MediaPlayer.exe executable remotely. There is only one problem: On Symbian, while we can pass command line arguments to MediaPlayer.exe, it just ignores them, so how do we tell it to open the input files produced by our fuzzer?

One solution is to place a small launcher application on the device that uses the CDocumentHandler::OpenFileL() API to launch our input files. This API looks at the filetype and launches the appropriate handler application. This approach also has the advantage that our fuzzer can automatically target the default registered applications for any given filetype.

Our launcher will be executed via the AppTRK channel, and takes one command line argument that contains the filename of our input file. Listing 9 shows the main function of the launcher application.

```
LOCAL_C void MainL() {

	CCommandLineArguments* args = CCommandLineArguments::NewLC();
	TInt nArgs = args->Count();

	if (nArgs != 2) {
		console->Printf(_L("missing filename parameter! Exiting\n"));
	} else {
		TPtrC filename(args->Arg(1));

		console->Printf(_L("trying to handle file: %S\n"), &filename);
```

```
            CDocumentHandler* handler = CDocumentHandler::NewL(NULL);
            CleanupStack::PushL(handler);

            TDataType emptyDataType = TDataType();

            handler->OpenFileL(filename, emptyDataType);
            }

      CleanupStack::PopAndDestroy(); // handler
      CleanupStack::PopAndDestroy(); // args
}
```

Listing 9: Target launcher application (Symbian C++)

The second problem we have to deal with is the occasional unexpected crash and restart of the device. To solve this problem we create another small application that we have automatically started at boot time. All this application does is wait for two minutes, and then attempt to start the AppTRK GUI, in an endless loop, using the RApaLsSession::StartApp() API (a new process will be started only if no instance of trkguiapp.exe is running). We also have to configure the phone to connect in PC Suite mode automatically. The code is shown in Listing 10.

```
LOCAL_C void MainL() {

      RProcess process;
      _LIT(KTrkPath, "");

      TThreadId app_threadid;
      CApaCommandLine* cmdLine;
      cmdLine=CApaCommandLine::NewLC();
      cmdLine->SetExecutableNameL(_L("C:\\sys\\bin\\trkguiapp.exe"));
      cmdLine->SetCommandL( EApaCommandRun );
      RApaLsSession ls;
      User::LeaveIfError(ls.Connect());

      while(1) {
            User::After(120000000);
            User::LeaveIfError(ls.StartApp(*cmdLine,app_threadid));
      }

      CleanupStack::PopAndDestroy(); // cmdLine
}
```

Listing 10: Target watcher application (Symbian C++)

Once we have installed our small helper apps on the phone, we can start writing the code that handles the upload and launching of files, and process debugging. We write a C++ program that does the following:

1. Connect to AppTRK via USB
2. Copy input file to device
3. Start the launcher app, passing it the name of the input file
4. Wait until launcher app has run successfully
5. Terminate launcher app
6. Search for our target application in the process list
7. Attach to the target process
8. Wait a short time for event notificiations
9. In case of an exception, write logfile with register values
10. Terminate the target process

The following code shows how we implemented this by using our MetroTRK DLL.

```
int handle_file_remote(char *sourcefile, char *targetfile, char *appname) {
      proclist_t proclist;             // list of processes
      thread_list_t threads;           // list of threads
      trk_process_info_t pi;                  // info about debugged process

      trk->set_notification_func(&handle_notification);

      if (debuglevel > 1) {
            trk->setdebug(true);
      }

      if (!(copy_file(sourcefile, targetfile, trk))) {
            printf("Failed to copy file to target device. Exiting\n");
            return 0;
      }

      if ( !(trk->create_process("C:\\sys\\bin\\launcher.exe", targetfile,
      &pi)) || !(trk->get_thread_list(pi.pid, &threads)))
      {
            printf("Failed to start launcher application. Exiting\n");
            return 0;
      }

      trk->resume_thread(pi.pid, threads.front().tid);

      for(int i = 0; i < 10; i ++) {
            trk->poll_for_event(1000);
      }

      trk->terminate_process(pi.pid);

      if ( !trk->get_process_list(proclist) )
      {
            printf("Failed to get list of running processes. Exiting\n");
            return 0;
      }

      int real_pid = 0;

      for ( int i=proclist.size()-1; i >= 0; i-- ) {
            if (strstr(proclist[i].name.c_str(), appname)) {
                  if (debuglevel) {
                        printf("Handler application found!\n",
proclist[i].name.c_str());
                        printf("%s\n", proclist[i].name.c_str());
                  }
                  real_pid = proclist[i].pid;
            }
      }

      if (!real_pid) {
            printf("Failed to find handler application\n");
            return 0;
      }

      if ( !trk->attach_process(real_pid)) {
            printf("Failed to attach to target process\n");
            return 0;
      }

      for(int i = 0; i < 100; i ++) {
```



```
                trk->poll_for_event(100);
        }

        trk->terminate_process(real_pid);
        printf("Process terminated.\n");

        return 0;
}
```

Listing 11: Remote file launcher and process debugger (C++ / WIN32)

Besides the code that copies and runs the file on the device, we need to register a notification handler. If we receive a TrkNotifyStopped notification, we have triggered an exception (in our case there is no other event that could have triggered this notification).

```
case TrkNotifyStopped:
{
        uint32 ea  = extract_int32(i, pkt);
        uint32 pid = extract_int32(i, pkt);
        uint32 tid = extract_int32(i, pkt);

        printf("Exception: pid = %d, tid = %d, pc=0x%08x\n", pid, tid, ea);
        printf("Description: %s\n", desc.c_str());
        printf("----------------------------------------\n", pid, tid, ea);

        uint32 rvals[17];

        trk->read_regs(pid, tid, 0, 17, rvals);

        for (int i = 0; i < 17; i++) {
                printf("%cR%d: %08X", i%4 == 0 ? '\n' : ' ', i, rvals[i]);
        }

        printf("\n");
        trk->terminate_process(pid);
}
```

Listing 12: Callback handler for the TrkNotifyStopped message (C++ / WIN32)

We will use a mutation based fuzzing approach. As templates for our fuzzed files, we have collected a set of small video files that are encoded with different video and audio codecs supported by Nokia smartphones. Our fuzzer mutates random sets of bytes, or bits, within these files[21]. A similar technique of media format fuzzing is described in (5).

For our fuzzing attempt we used a set of five input files which are listed in table 1.

Filename	File Size	Container	Video Codec	Audio Codec
01_3gp_h263_AMR.3gp	159 KB	MPEG4 (3GPP Media Release 4)	H.263	Adaptive Multi-Rate
02_3gp_mp4_mp2.3gp	61.1 KB	MPEG4 (3GPP Media Release 4)	MPEG-4 Visual	Advanced Audio Codec Version 4
03_mp4_h264_mp2.m4v	157 KB	MPEG-4	Advanced Video Codec Baseline@L1.3	Advanced Audio Codec Version 4
04_realvideo.rmvb	128 KB	RealMedia	RealVideo 4 RV40 Based on AVC (H.264), Real Player 9	Cooker Based on G.722.1, Real Player 6
05_wmv_wm9.wmv	22.2 KB	Windows Media	VC-1 Windows Media Video 9	Windows Media Audio 9.2

Table 1: Fuzzer input files

The following mutations were applied to each file:

- 32 random byte mutations (2048 mutations per file min.)
- 64 random byte mutations (2048 mutations per file min.)
- 64 random bit flips (2048 mutations per file min.)

The fuzzing target was RealPlayer S60 HX cvs_cays_221 20080825 running on our Nokia N96.

[21] Actually, I wrote a second fuzzer that used hachoir to dissect the media files, and was able to create targeted mutations of integer fields and character blocks. But as it turned out, random byte and bit fuzzing found bugs much faster, so in the end I discarded the 'advanced' version.

Figure 11: The fuzzer running (longcat was inserted to hide PC addresses)

FINDINGS ANALYSIS

As a result of our fuzzing attempt, we found that MediaPlayer.exe would crash in various ways for all input files, except for the WMV / Windows Media encoded video file. While many of the exceptions were user panics, some of them were data abort exceptions, resulting from attempted reads, writes or executes of invalid memory addresses.

FUZZING RESULTS

The following table contains the list of the results. The following assumptions were made when generating the list:

1. Data abort exceptions with a different PC were treated as separate bugs. This is not a precise method, since the same root cause may lead to data aborts at different code addresses. For this reason, the list of data abort exceptions may contain duplicates.
2. When a user panic exception is thrown, the program counter always points into euser.dll. In this case, the values in registers R1 and R3 were inspected to eliminate duplicates. However, for the same reason as above, not all duplicate bugs may have been eliminated.

Because the bugs are unfixed at the time of publication of this paper[22], we have removed the code addresses of all data abort exceptions from the result table. Additionally, we will not include information about the byte positions or fields mutated within the files.

[22] The results have been sent to Nokia and are under review at the time of this writing.

ID	Input file	Fuzzing mode	Exception Type	PC	Module
01e4c0b3	02 (MP4/MP2)	random byte	User Panic 23	0xf8556e5f	euser.dll
06b6d67a	02 (MP4/MP2)	random byte	User Panic 23	0xf8556e5f	euser.dll
072fd6ab	**01 (h.263/AMR)**	**bit flip**	**Data abort**	**[REMOVED]**	**rarender.dll**
08db253b	04 (RealVideo)	random byte	User Panic 45	0xf8556e5f	euser.dll
09a10702	02 (MP4/MP2)	bit flip	User Panic 23	0xf8556e5f	euser.dll
19dc61c1	**04 (RealVideo)**	**random byte**	**Data abort**	**[REMOVED]**	**euser.dll**
270334f8	**03 (h.264/mp2)**	**random byte**	**Data abort**	**[REMOVED]**	**STH264DecHwDevice.dll**
287d7bec	02 (MP4/MP2)	random byte	User Panic 23	0xf8556e5f	euser.dll
2b1be124	**03 (h.264/mp2)**	**random byte**	**Data abort**	**[REMOVED]**	**STH264DecHwDevice.dll**
2e973d62	**04 (RealVideo)**	**random byte**	**Data abort**	**[REMOVED]**	**clntcore.dll**
2f3eca69	**04 (RealVideo)**	**bit flip**	**Data abort**	**[REMOVED]**	**clntcore.dll**
365b6ddf	02 (MP4/MP2)	random byte	User Panic 23	0xf8556e5f	euser.dll
3eb2d38f	**04 (RealVideo)**	**random byte**	**Data abort**	**[REMOVED] [REMOVED]**	**euser.dll HxMmfCtrl.dll**
47267718	03 (h.264/mp2)	bit flip	User Panic 23	0xf8556e5f	euser.dll
684ca479	03 (h.264/mp2)	random byte	User Panic 23	0xf8556e5f	euser.dll
68551fd5	**04 (RealVideo)**	**random byte**	**Data abort**	**[REMOVED]**	**HxMmfCtrl.dll**
6b3b3b34	03 (M4V)	random byte	User Panic 23	0xf8556e5f	euser.dll
7cd6c505	02 (MP4/MP2)	bit flip	User Panic 23	0xf8556e5f	euser.dll
7d6ac7f4	**03 (h.264/mp2)**	**random byte**	**Data abort**	**[REMOVED]**	**mdfh264payloadformat.dll**
88c573bc	**04 (RealVideo)**	**random byte**	**Data abort**	**[REMOVED]**	**mdfvidrender.dll**
92c4be96	02 (MP4/MP2)	random byte	User Panic 23	0xf8556e5f	euser.dll
937b5e94	**03 (h.264/mp2)**	**bit flip**	**Data abort**	**[REMOVED]**	**euser.dll**
9e490a1f	**04 (RealVideo)**	**random byte**	**Data abort**	**[REMOVED] [REMOVED]**	**mdfvidrender.dll MMFDevSound.dll**
a5c2ff10	**04 (RealVideo)**	**random byte**	**Data abort**	**[REMOVED]**	**clntcore.dll**
af2acdb7	02 (MP4/MP2)	bit flip	User Panic 23	0xf8556e5f	euser.dll
b44110c3	03 (M4V)	random byte	User Panic 23	0xf8556e5f	euser.dll
cf2674ef	03 (h.264/mp2)	bit flip	User Panic 23	0xf8556e5f	euser.dll
d1c130c4	02 (MP4/MP2)	bit flip	User Panic 23	0xf8556e5f	euser.dll
d3c390bc	02 (MP4/MP2)	random byte	User Panic 23	0xf8556e5f	euser.dll
d5c18df2	**03 (h.264/mp2)**	**bit flip**	**Data abort**	**[REMOVED]**	**mdfh264payloadformat.dll**
d82f7cec	02 (MP4/MP2)	random byte	User Panic 23	0xf8556e5f	euser.dll
dc9923ac	03 (M4V)	random byte	User Panic 23	0xf8556e5f	euser.dll
dee1235d	**03 (h.264/mp2)**	**bit flip**	**Data abort**	**[REMOVED]**	**euser.dll**
df915a5d	03(MP4)	random byte	User Panic 23	0xf8556e5f	euser.dll
eb2aa88a	**04 (RealVideo)**	**random byte**	**Data abort**	**[REMOVED] [REMOVED]**	**HxMmfCtrl.dll ArmRV89Codec.dll**
ef91d53b	02 (MP4/MP2)	random byte	User Panic 23	0xf8556e5f	euser.dll
f473afd9	03 (h.264/mp2)	random byte	User Panic 23	0xf8556e5f	euser.dll

Table 2: Fuzzing results

The results listed in this table apply to the Nokia N96 smartphone with firmware version 11.018. We also played the resulting files on a Nokia E61i and a Nokia E71 with varying results, with some of the files hanging or crashing the RealPlayer process or rebooting the phones, which seems to indicate that at least some of the vulnerabilities may affect other Nokia smartphone models than the N96. This is to be expected, since the multimedia software installed on other Nokia phones is similar to that of the N96. However this has not been tested systematically.

DETERMINING EXPLOITABILITY OF SPECIFIC BUGS

The last question we want to answer is whether any of these bugs may allow for manipulation of the program counter, and therefore execution of arbitrary code, within the exploited process.

We can forget about the user panic exceptions right away, because these are most likely not exploitable[23]. For example, a user panic 23 is thrown when the process attempts to write data into a string descriptor of insufficient size (the Symbian equivalent of a buffer overflow). But this is detected by the descriptor implementation before it happens, and no memory corruption occurs.

What remains are the 14 detected data abort exceptions. To find out if a particular bug is exploitable, the first thing we will be looking at is the code that triggers the exception. Let us have a look at exception no. #19dc61c1 found by our fuzzer. It is an abort that happens at address [REMOVED]. Our symbol database tells us that this address belongs to euser.dll.

Disassembling euser.dll at this address shows the following code:

```
[REMOVED]  LDR     R3, [R3,#0x10]
[REMOVED]  STR     R12, [R0,#8]
[REMOVED]  STR     R0, [R7]
[REMOVED]  BX      R3
```

Listing 13: Code inside euser.dll triggering the data abort exception (ARM ASM)

Initially, register R3 seems to point to some structure in memory that contains a function pointer at offset +16. This function pointer is loaded into R3, and is called a few instructions later. The exception happens when the function pointer is loaded into R3 at [REMOVED].

An inspection of the fuzzer logfile tells us the contents of the registers at this point (listing 14).

```
Exception: pid = 457, tid = 477, pc=[REMOVED]
Description: A data abort exception has occurred.
-------------------------------------------------
R0: 008ED5A4      R1: 00000000      R2: 00000000      R3: 080F1217
R4: 008000F8      R5: 0040AE8C      R6: F8550ACD      R7: 0040AE60
R8: 00000000      R9: 00000040      R10: 6422E440     R11: 00000000
R12: 958A826C     R13: 0040AE40     R14: [REMOVED]    R15: [REMOVED]
```

Listing 14: Register values at the time of the crash

R3 is pointing to an invalid memory location, which is why the data abort exception occurs.

[23] It may actually happen that a user panic is caused by some kind of precedent, undetected memory corruption. But this is rather unlikely, and we will ignore the possibility in our further analysis.

At this point, we could start a detailed manual analysis of the root cause of the crash by means of static analysis, which would be the most thorough way to determine its exploitability. This is however the most costly and strenuous approach which I only like to use as a last resort. A quicker way to do this is to search for a way to directly or indirectly manipulate the register we need to change. The approach we used to do this is the following:

1. Use the fuzzing engine to isolate, by means of binary search, the position(s) or field(s) within the input file that are responsible for the error condition
2. Use the output of step 1 as a template for further random fuzzing with the goal of changing the value of the target register
3. In case a fuzzing attempt successfully changes the register, isolate the responsible position(s) inside the input file

In this specific case, it turned out to be possible to change the value inside the target register (R3) by further manipulating the input file. The following shows the crash dump with R3 being pointed to ASCII 'AAAA' (41414141):

```
Exception: pid = 7841, tid = 7849, pc=
Description: A data abort exception has occurred.
----------------------------------------

R0: 008ED5A4      R1: 00000000      R2: 41433F43      R3: 41414141
R4: 008000F8      R5: 0040AE8C      R6: F8550ACD      R7: 0040AE60
R8: 00000000      R9: 00000040      R10: 641BB640     R11: 00000000
R12: 41414140     R13: 0040AE40     R14:              R15:
```

Listing 15: Register values with R3 manipulated to value 0x41414141

This should be sufficient to denote this condition as exploitable, since an attacker could point the register to a value placed on the stack or heap (e.g. inside the input file itself) and have this value loaded into the program counter register.

CONCLUSION

In this paper, we have shown how to identify low level vulnerabilities in programs running from the ROM of Symbian devices. Finding and debugging bugs may be more difficult than on other smartphone platforms, but once the proper toolchain is in place, standard methods like input fuzzing can be applied and vulnerabilities can be identified with relative ease.

Mobile phone manufacturers should be aware that remote vulnerabilities of the kind discussed in this paper could be used in targeted attacks to remotely compromise a smartphone, or as a means of propagation for a mobile network worm. In a worst case scenario, such a worm may replicate over the mobile network, which would, besides probably bringing down the network, cause massive financial damage.

RECOMMENDATIONS

Bugs like those discussed in this paper can be prevented by introducing rigid software quality assurance. Phone manufacturers should make sure that the mobile operating systems and applications shipped on their devices are developed with secure development practices in mind, and should apply thorough security testing to all applications before integrating them into the firmware images.

One essential security feature that is missing from Symbian OS based smartphones, or at least from the ones running S60, is automatic firmware updates. While it is possible to perform a manual firmware upgrade, users are not encouraged to do so, and the process is too complicated for the average end user to bother. As a result, a Symbian OS smartphone often runs its original firmware over its entire lifetime. This means that even if security bugs are fixed, most of the phones out there will remain unpatched! To improve on this situation, manufacturers of Symbian OS smartphones should implement an automatic update mechanism and encourage users to quickly apply security patches as they become available.

To summarize, an improved software quality management process, combined with automatic updates to rapidly fix known flaws and a push of end user awareness, would do a lot to help mitigate security risks that may arise for customers. These risks may increase should vulnerability research on Symbian OS finally take off, and should vulnerabilities be found and published regularly. Properly implemented, these measures would be more effective than the currently used platform security approach, which in fact creates ‘security by obscurity’ by placing restrictions on software and end user devices.

From an end user perspective, security best practices should be applied that are similar to those required on desktop PCs. The following list contains some of the most important guidelines:

- Perform regular software updates
- Do not install unnecessary applications and services
- Use Anti Virus software
- Take care when browsing the web
- Do not open SMS, MMS or emails from unknown sources[24]

[24] This would not be sufficient if there were an SMS or MMS worm sending itself to contacts taken from an infected phone’s addressbook. In this case the only rule that would help would be not to open anything at all.

SYMBIAN/ARM SPECIFIC EXPLOITATION TECHNIQUES

We are currently lacking the elegant exploitation techniques available for other architectures. How do we efficiently exploit overflows on the stack and heap? Where do we find global function pointers to overwrite? Which memory regions could be used for trampolines or jump addresses? What about the NX protections on ARMv6 architectures? These questions will be answered in a subsequent research project.

SYMBIAN SHELLCODE

The essentials of shellcoding on Symbian/ARM have been discussed in (1), but there's still a lot of room for improvement. One drawback with the shellcode proposed in (1) is the method used to elevate the privileges of the exploited process, which is done by having the shellcode access the Symbian Signed website via the web browser and sign a SIS package specifically for the target device. While this could probably work, it would require a very large and complex shellcode, and would only obtain a subset of the available capabilities.

Let's remember our discussion of platform security. What would stop us from building a platform security exploit into our shellcode? This way, our shellcode could launch a second stage payload with arbitrary capabilities.

A shellcode using the mapdrives exploit could for example do the following

1. Call RFs::MkDirAll('E:\\x\\sys\\bin\\')
2. Write second stage to file E:\\x\\sys\\bin\\x.exe
3. Map to drive with RFs::SetSubst()
4. Call RProcess::Create('x.exe')

This would allow the second stage executable to do practically anything on the device.

The obvious drawback of this idea is that for the shellcode to work, the exploited process needs to have the DiskAdmin capability. However this is not too uncommon for executables found in the firmware. Luckily, when planning our exploit, we know in advance which capabilities are available to our target process.

The following Perl script can be used to extract the capabilities set from the header of a given ROM image:

```
#!/usr/bin/perl

my @caps = ("TCB",
        "CommDD",
        "PowerMgmt",
        "MultimediaDD",
        "ReadDeviceData",
        "WriteDeviceData",
        "DRM",
        "TrustedUI",
        "ProtServ",
        "DiskAdmin",
        "NetworkControl",
        "AllFiles",
        "SwEvent",
```

```
        "NetworkServices",
        "LocalServices",
        "ReadUserData",
        "WriteUserData",
        "Location",
        "SurroundingsDD",
        "UserEnvironment");

$fn = shift @ARGV;

open(F, $fn) || die "could not open input file! $! \n";
seek(F, 0x4C, SEEK_SET);
read(F, $buf, 4);
close(F);

my ($caps) = unpack("L", $buf);

$nbit = 0x1;

for ($i = 0; $i < 20; $i ++) {
       if ($caps & $nbit) {
              print @caps[$i]."+";
       }

       $nbit <<= 1;
}
```

Listing 16: Displaying capabilities of a ROM image file (Perl)

Now we can check the capabilities available to the MediaPlayer executable (our fuzzing target from the last chapter). The output is shown in listing 17.

```
$ perl getcap.pl ZBIN/MediaPlayer.exe
MultimediaDD
ReadDeviceData
WriteDeviceData
DRM
DiskAdmin
SwEvent
NetworkServices
LocalServices
ReadUserData
WriteUserData
Location
UserEnvironment
```

Listing 17: Capabilities available to MediaPlayer.exe

As it turns out, MediaPlayer.exe does, for some reason, actually have the DiskAdmin capability! If we were to exploit MediaPlayer.exe, the proposed shellcode would work.

There are however many other executables that do not own the required capabilities. For these, a different platform security exploit is needed. This, and Symbian shellcode in general, is an interesting topic for research and one we hope to discuss in a future paper.

SYMBIAN ROOTKITS

Once the exploitation and shellcode techniques are in place, we need something to do with the compromised smartphones. If we really can obtain TCB capabilities with our shellcode, the possibilities are practically endless! We can abuse all of the smartphone's features for our purposes. Some interesting features for a rootkit would include:

- Stealing SMS, contacts, emails, and other data from the phone
- Recording and hijacking phone conversations
- Recording a video stream from the smartphone's camera
- Tracking the location of the smartphone's owner
- Placing calls to pay numbers
- Creating a botnet of smartphones

Another interesting area of research are file- and process hiding techniques, which are needed to keep a rootkit hidden from the user.

NOKIASTALKER IRC BOT

As a proof of concept, we have written an IRC bot trojan that connects the smartphone to a preset IRC server. The "infected" phone can then be controlled with commands sent via private chat. Figure 12 shows the bot being used to dump the contact list of a phone connected to the IRC server.

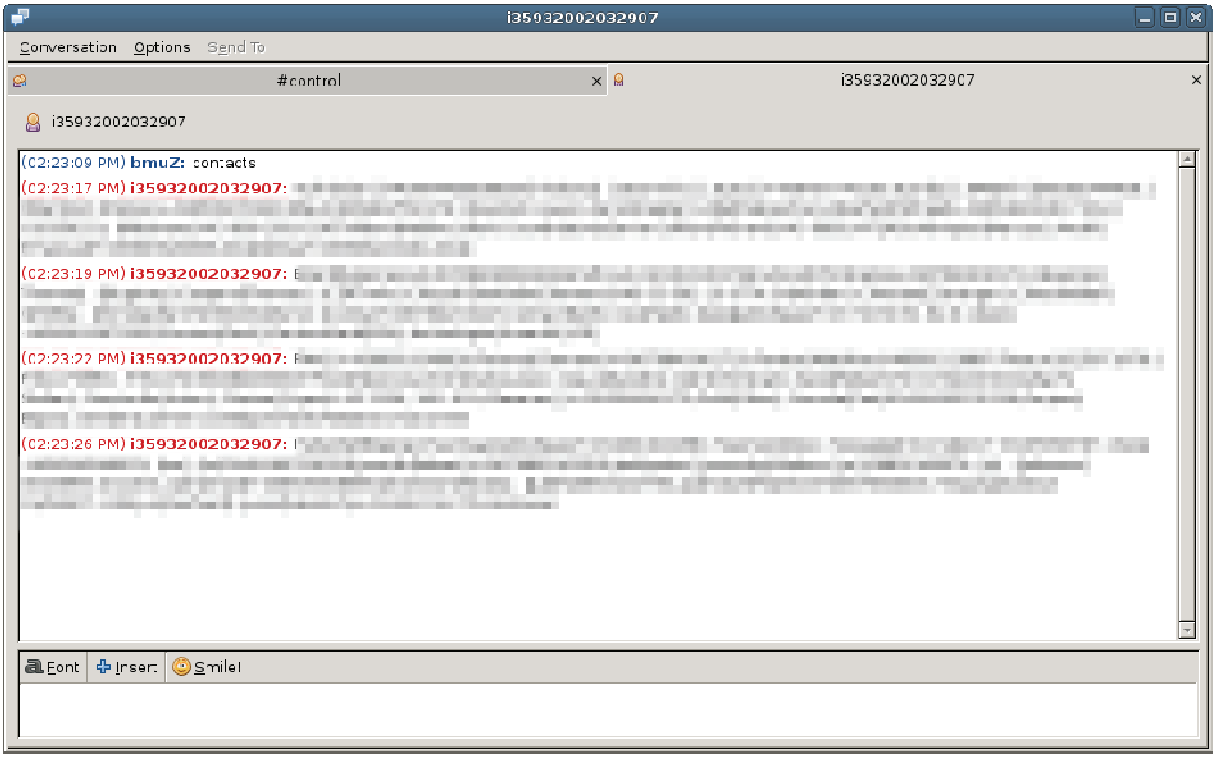

Figure 12: Dumping contacts of a compromised smartphone

The bot is based on the OpenC IRC example implementation, which is available on the Forum Nokia website[25]. It is intended only for demonstration purposes and will be kept private for now.

SYMBIAN OS IS GOING OPEN SOURCE!

In 2008, Nokia acquired Symbian Ltd. and formed the Symbian Foundation together with other partners. The Symbian Foundation has announced plans to unify Symbian OS, S60, UIQ and

[25] http://www.forum.nokia.com/Tools_Docs_and_Code/Code_Examples/Open_C_and_C++.xhtml

MOAP(S) to create a single open source platform. The Symbian OS source code is expected to be released under the Eclipse license. According to the release plan on the Foundation's blog, the first devices based on the new open source platform could be released at the end of 2009[26]. It will be interesting to see how this will affect the possibilities available to security researchers interested in Symbian OS.

For one, it will be possible to review the code of the core OS when it becomes available. The Symbian Signed / capabilities system probably won't be changed, and phone manufacturers will continue to sell locked down phones. But with the kernel and system server source code available, it should be much easier to build testing toolchains that are more effective than the one described in this paper. We will wait and see how things turn out.

ACKNOWLEDGEMENTS

The author would like to thank Ilfak Guilfanov from Hex Rays, for offering great support, providing patches for the IDA Pro Symbian debugger plugin in real time, and permitting the reuse of code contained in the IDA SDK for this project (and of course for writing IDA Pro in the first place).

Many thanks also go to the guys at the www.symbian-freak.com modding forum (FCA00000, Zorn, bugb, and many others) – the source of the platform security hack used in this paper, as well as many useful tools and forum threads.

Last but not least I would like to thank David Matscheko of SEC Consult for ideas and help with the media file fuzzer, and David Niedermaier, David White and Johannes Greil for proof-reading and technical reviews.

Also thanks to the observant readers Marco Bellino and Gunther for pointing out wrong references and credits.

REFERENCES

1. **Mulliner, Collin.** *Exploiting Symbian: Symbian Exploitation and Shellcode Development.* [Online] 2008. http://www.blackhat.com/presentations/bh-jp-08/bh-jp-08-Mulliner/BlackHat-Japan-08-Mulliner-Hacking-Symbian-OS.pdf.

2. **(ARTeam), Shub Nigurrath.** Primer on Reversing Symbian S60 Applications. [Online] http://news.nopcode.org/pdf/Primer_on_Reversing_Symbian_S60_Applications_by_Shub-Nigurrath_v14.pdf.

3. **Sales, Jane.** *Symbian OS Internals: Real-time Kernel Programming.* s.l. : John Wiley & Sons, Ltd, 2005. ISBN-13 978-0-470-02524-6.

4. **Freescale Semiconductor.** *CodeWarrior MetroTRK Reference.* [Online] Freescale Semiconductor Inc., 2004. http://www.freescale.com/files/soft_dev_tools/doc/ref_manual/METROTRKRM.pdf.

[26] http://blog.symbian.org/2009/03/12/introducing-the-release-plan/

5. **Thiel, David.** Exposing Vulnerabilities in Media Software. [Online] 08 02, 2007. http://www.isecpartners.com/files/Blackhat_2007_Thiel_Exposing_Vulnerabilities_Media_Software.pdf.

6. **Stichbury, Jo.** *Symbian OS Explained: Effective C++ Programming for Smartphones.* s.l. : John Wiley & Sons Ltd., 2005. ISBN 0-470-02130-6.

ABOUT THE AUTHOR

Bernhard Müller is a senior security analyst at SEC Consult Unternehmensberatung GmbH. He has been doing research on various information security topics for several years, and has found and reported vulnerabilities in products of companies such as Nortel, Macromedia and Microsoft. It said that he figured out the 2008 DNS flaw within less than an hour, by quickly skimming over the RFC. He has also posted a security advisory about a cross site scripting flaw once, which is repeatedly pointed out to him by some colleagues.

ABOUT THE SEC CONSULT VULNERABILITY LAB

Members of the SEC Consult Vulnerability Lab perform security research in various topics of technical information security. Projects include vulnerability research and the development of cutting edge security tools and methodologies, and are supported by partners like the Technical University of Vienna. The lab has published security vulnerabilities in many high-profile software products, and selected work has been presented at top security conferences like Blackhat and DeepSec.

For more information, see http://www.sec-consult.com/

For feedback or questions, please contact:

research [at] sec-consult [dot] com