Embedded Reverse Engineering: Cracking Mobile Binaries

1. Overview

Reverse-engineering has long been one of the most popular trouble shooting techniques. In fact, long before the first hacker ever laid eyes on a computer screen, technicians, engineers, and even hobbyists were busy tearing apart mechanical devices to see if they could deduce their seemingly magical operations with the hopes of making it work better, or at the very least, hoping they could understanding what made a device tick. Over the years, this concept has been passed on to the computer profession, where the concept of reverse-engineering evolved into one of the most powerful methods of learning available. Ironically, this very useful technique has fallen under attack and is being threatened by various nefarious Acts and policy control groups.

If a computer professional has been in the field for any length of time, they have already used reverse-engineering to their benefit. In fact, the open-source community uses reverse-engineering as one of their main tools for learning software and figuring out what a program does, or in some cases, doesn't do. However, there is one major branch of computing that has had little headway in the arena of reverse-engineering. This elusive niche is the PocketPC application.

To help fill this gap, and to increase the awareness of PocketPC reverse-engineering, this paper/discussion will provide an overview of what is required, and how one can reverse their PocketPC. The following pages will provide an overview of the PocketPC environment, the tools required to successfully reverse-engineering Windows CE, and the methods by which a person can dig deep inside an application to alter code as they see fit.

Note, this article/discussion will skirt the borders of many ethical and moral issues. The information in this paper is presented from a researchers point of view for educational purposes only. We firmly believe that when a product is purchased, be it a can of soup or software program, the owner should be able to do with it as they please, with the arguable exception of manipulative EULA in which the software is rented. Please note, the information presented is not meant to promote the theft of software.

2. Windows CE Architecture

Windows CE is the operating system of choice for most pocket PC devices. As such, it is important to understand the basics of how this operating system works to become proficient at reverse engineering on the PPC platform. This segment of the paper will outline the particulars of Windows CE, and what it means to you when researching the characteristics of a program. Note, this segment will only briefly cover the Windows CE architecture, with some deeper looks at sections important to understand when reverse-engineering a program. For more information about this subject, the Microsoft.com provides a wealth of information. Please note that much of this information can be applied to any Windows OS; therefore, please feel free to jump ahead if you are familiar with this subject.

2.1 Processors

In this world of miniature gadgets, only so much is possible. Physical properties often determine how far technology can go. In the case of the pocket PC's this is also true. Heat generated by high-speed processors in notebook PC's have been known to burn people and even has provided enough heat to fry eggs. If the same processor were used in a pocket PC, a user would have to wear hot pads to operate just to hold it.

As a result, Windows CE devices are limited in their choice of processors. The following is the list of processors supported by Windows CE.

- **ARM:** Supported processors include ARM720T, ARM920T, ARM1020T, StrongARM, XScale
- **MIPS:** Supported processors include MIPS II/32 w/FP, MIPS II/32 w/o FP, MIPS16, MIPS IV/64 w/FP, MIPS IV/64 w/o FP
- **SHx:** Supported processors include SH-3, SH-3 DSP, SH-4
- **x86:** Supported processors include 486, 586, Geode, Pentium I/II/III/IV

If heat dissipation is a serious issue, the best choice is one of the non-x86 processors that use a reduced level of power. The reduction in power consumption will reduce the amount of heat that is created during processor operation, but also limits the processor speed.

2.2 Kernel, Processes, and Threads

The follow section will describe the core of the Windows CE operating system, and how it processes information.

2.2.1 Kernel

The kernel is the key component of a Windows CE device. It handles all the core functions of the OS, such as process, thread and memory management. In addition, it also handles scheduling and interrupt handling. However, it is important to understand that Windows CE used parts of the desktop Windows software. This means it has a similar threading, processing, and virtual memory model as the other Windows OSes.

While the similarities are undeniable, there are several items that make this OS a completely different beast. These center on the use of memory and the simple fact that there is no hard drive (discussed later in the Memory Architecture section). In addition, DLLs in Windows CE are not implemented as they are in other Windows operating systems. Instead, they are used in such a way as to maximize the amount of available memory. By integrating them into the core operating system, DLLs don't take up precious space when they are executed. This is an important concept to understand before attempting to RVE a program in Windows CE. Due to this small difference, attempting to break a program while it is executing a system DLL is not allowed by Microsoft's EVT (MVT).

2.2.2 Processes

A process in Windows CE represents an executing program. The number of processes is limited to 32, but each process can execute a theoretically unlimited number of threads. Each thread has a 64k memory block assigned to it, an ID, and a set of registers. It is important to understand this concept because when debugging a program, you will be monitoring the execution of a particular thread, its registers, and the allotted memory space. By doing this, you will be able to deduce hidden passwords, serial numbers, and more.

Processes can run in two modes; kernel and user. A kernel process has direct access to the OS and the hardware. This gives it more power, but a crash in a kernel process will often crash the whole OS. A user process, on the other hand, operates outside the kernel memory, but a crash will only kill the running program, not the whole OS. In Windows CE, any 3rd party program will operate in user mode, which means it is protected. In other words, if you crash a program while RVEing it, the whole OS wont crash (though you still may need to soft boot the device).

There are two other points that should be understood. First, one process cannot affect another processes data. While related threads can interact with each other, a process is restricted to its own memory slot. The second point to remember is that each existing thread is continuously being stopped and restarted by a scheduler (discussed next). This is how multitasking is actually performed. While it may appear that more than one program is running at a time, the truth remains that only one thread may execute at any one time.

2.3 Scheduler

The Scheduler is responsible for managing the thread process time. It does this by giving each thread a chance to use the processor. By continuously moving from thread to thread, the scheduler ensures that each gets a turn. Built into the scheduler are three important features that are important to understand.

The first feature is a method that is used to increase the amount of processor time. The secret is found in multi-threading an application. Since the Scheduler assigns processor time at the *thread* level, a process with 10 threads will get ten times the processor time than a process with one thread.

Another method in gaining more processor time is to increase the process priority. However, this is not encouraged unless necessary. Changing priority levels can cause serious problems in other programs, and will affect the speed of the computer as a whole. One priority that needs to be mentioned is the THREAD_PRIORITY_TIME_CRITICAL priority that forces the processor to complete the critical thread until it is complete.

The final interesting fact deals with a problem that can arise when priority threading is used. If a low priority thread is executing and it ties up a resource needed by a higher priority thread, the system could become instable. In short, this creates a paradox where the high thread will wait for the low thread to finish, which is in turn waiting on the high to complete. To prevent this from occurring, the scheduler will detect such a paradox and boost the lower priorities thread to a higher level allowing it to finish.

2.4 Memory Architecture

One of the most obvious properties of a device running Windows CE is that it doesn't have a hard drive. Instead of spinning disks, pocket PC's use old fashion RAM (Random Access Memory) and ROM (Read Only Memory) to store data. While this may seem like a step back in technology, the use of static memory, like ROM, is on the rise and will eventually make moving storage devices obsolete. The next few paragraphs will explain how memory is used in a Windows CE device to facilitate program execution and use.

In a Windows CE device, the entire operating system is stored in ROM. This type of memory is typically only read from and is not used to store temporary data that can be deleted. On the other hand, data in RAM is constantly being updated and changed. This memory is used to hold all files and programs that are loaded into the Windows CE-based device, as well as the registry and various data files required by CE applications.

RAM not only stores data, but it is also used to execute programs. When a 3rd party program is executed, it is first uncompressed, then copied into another part of RAM, and executed from there. This is why having a surplus of RAM is important in a Windows CE device. However, the real importance of RAM is found in the fact that its data can be written to and accessed by an address. This is necessary because a program will often have to move data around. Since a program is allotted a section of RAM to run in when it is executed, it must be able to write directly to its predefined area.

While ROM is typically only used as a static storage area, in Windows CE it can be used to execute programs, which is know as Execute In Place (XIP). In other words, RAM won't be required to hold the ROMs data as a program executes. This allows RAM to be used for other important applications. However, this only works with ROM data that is not compressed. While compression will allows more data to be stored in ROM, the decompression will force any execution to be done via the RAM.

RAM in a Windows CE device is split between two functions. The first is object store, which is used to hold files/data that is used by the programs, but is not stored in the ROM. In particular, the object store holds compressed program/user files, database files that hold structured data, and the infamous Windows registry file. Though this data is stored in RAM, it remains intact when the device is 'turned off'. This is due to the fact that the RAM is kept charged by the power supply, which is why it is very important to never ever let the charge on a pocket PC completely die. If this happens, the RAM will loose power and will reset. This will then dump all installed programs and will basically wipe everything on the device except for what is stored in ROM. This is also referred to as a hard boot when dealing with a pocket PC device.

The second function of the RAM is to facilitate program execution. As previously mentioned, when a program is running it needs to store information it is using. This is the same function that RAM serves on a typical desktop PC. However, this also means that any data passing through a program, such as a password or serial number, will be written to the RAM at one time or another.

Windows CE does have a limit on the RAM size. In Windows CE 3.0 it is 256 MB with a 32 MB limit on each file, but in Windows CE .NET this value has been increased to a rather large 4GB. In addition, there is a limit to the number of files that can be stored in RAM of 4,000,000. There are other limits, such as the number of programs that can operated at the same time, which brings us to multitasking.

Windows CE was designed to be a multitasking operating system. Just like other Windows operating systems, this is important to allow more than one program to be open at a time. In other words, you can listen to an MP3 while taking notes, and checking out sites on the Internet. Without multitasking, you would be forced to close one program before opening another. However, you must be careful opening to many programs in a Windows CE device. Since you are limited by the amount RAM in the device and each open program takes up a chunk of the RAM, you can quickly run out of space.

Finally, the limitation of RAM in a pocket PC also has impacted the choice of operating system. Since Windows CE devices only have 32-128 MB of internal RAM, they don't make good platforms for operating systems that use a lot of memory, such as Embedded Windows XP. In this OS, the minimum footprint for a program is 5MB. On the other hand, Windows CE only requires 200k; this is a 2500% difference. When RAM is limited by space and pricing considerations, the affects are far reaching.

2.5 Graphics, Windowing and Event Subsystem (GWES)

This part of the Windows CE architecture is responsible for handling all the input (e.g. stylus) and output (e.g. screen text and images). Since every program uses windows to receive messages, this is a very important and key part of Windows CE. As a result, this is also one of the key areas you need to understand to successfully RVE a program.

Without going into too much detail, you should know that every Windows CE process created when a program executes is assigned its own windows messaging queue. This queue is similar to a stack of papers, which is added to and read from. This queue is created when the program calls GetMessage, which is very common in Windows CE programs. While the program executes and interacts with the user, messages are placed on and removed from the queue. The following is a list and explanation of the common commands that you will see while RVE.

PostMessage
Places message on queue of target thread, which is returned immediately to the process/thread.

SendMessage
Places message on queue, but does not return until it is processed.

|SendThreadMessage
Sends messages directly to thread instead of the queue

These Message commands, and others, act as virtual flares when RVE a program. For example, if a "Sorry, wrong serial number" warning is flashed to the screen, you can bet that some Message command is used. Therefore, by looking for the use of this command in a disassembler, you can find the part of the program that needs further research.

2.6 Summary

The last few pages have given you an inside look at how Windows CE operates. This information is required reading for the rest of this paper. In fact, by understanding how a processor deals with threads, the memory architecture, and how Windows CE uses messages to communicate with the executing program, you will have an easier time understanding how RVE works. Just as a doctor must understand the human body before troubleshooting even a head ache, a RVE must understand the platform they are dissecting if they going to be successful at making at patch or deciphering a serial number.

3 Reverse Engineering Fundamentals

3.1 Overview

When a developer writes a program, they typically use one of several languages. These typically include Visual Basic, C++, Java or any one of the other lesser used languages.

The choice of language depends on several factors. The most common being space and speed considerations. In the infamously bloated Windows environment, Visual Basic is arguable the king. This is because the hardware required to run Windows is usually more than enough to run any Visual Basic application. However, if the programmer needs a higher level of speed and power, they will probably select C++.

While these upper level languages make programming easier by providing a whole selection of Application Program Interfaces (API) and commands that are easy to understand, there are many occasions where a programmer must create a program that can fit in a very small amount of memory, and operate extremely quickly. To meet this goal, they will choose to use a language known as Assembler. This low level language allows a coder to write directly to the processor, thus controlling the hardware of the computer directly. However, programming in assembler is very tedious and must be done within a very explicit set of rules.

As we have hinted, programming languages exist on several different levels. The lowest level languages speak right to the hardware, and typically require little in the way of conversion. On the other hand, upper level languages like VB and SQL are often easy to write. However, these languages must be compiled one or more times before the instructions can be understood by the hardware responsible for executing it. In fact, many of these upper level languages don't really have any way of controlling hardware, but must make calls to other files and programs that can make hardware calls in proxy.

Without going to deep into the nuances of programming languages, the point of this discussion is to ensure that you understand that almost every program will end up as assembler code. Due to this, if you really want to have control over a computer and the programs on the computer, you must understand assembler code. Since each and every processor type uses its own set of assembler instruction, you need to focus on one device (i.e. one processor type) and become fluent in the operation codes (opcodes), instruction sets, processor design, and how the processor uses internal memory to read and write to RAM. It is only after you have mastered the basics of the processor operation that you can start to reverse-engineer a program. Fortunately, most processors operate very similar to each other, with slight variations in syntax and use of internal processor memory.

Since our target processor is the ARM processor used by PDA's, we will provide some of the necessary information you need to know, or at least be familiar with, before attempting to study a program meant to run on this processor type. The next few pages will provide you with a description of the ARM processor, its major op codes, their HEX equivalent, and how the memory is used. If you do not understand this information, you may have some difficulty in following the rest of this paper.

3.2 Hex vs. Binary

To successfully RE a program, there are several concepts that you must understand. The first is that no matter what programming language a file is written in, it will eventually be converted to a language that the computer can understand. This language is known as binary and exists in a state of ones and zeros. For example, the word "HACKER" in binary is written as follows:

H	A	C	K	E	R
01001000	**01000001**	**01000011**	**01001011**	**01000101**	**01010010**

While people did code in binary at one time, this is very rare in today's interface based world. In fact, many operating systems do not display, store, or even transmit this binary information, as it really exists; instead, they use a format known as HEX.

Hex, while still very cryptic, shortens the process of transmitting data by converting the 8 digit binary byte, into a 2 character hex value. For example, the previously illustrated word "HACKER" in binary would equate to the following in hex:

H	A	C	K	E	R
48	**41**	**43**	**4B**	**45**	**42**

In addition to the space considerations, experienced computer programmers can easily understand hex characters. In fact, with nothing more than a simple hex editor, a knowledgeable hacker can open an executable file and alter the hex code of the file to remove protection, alter a programs appearance, or even install a Trojan. In other words, understanding hex is one of the main requirements of being able to reverse-engineer a program. To facilitate you in your endeavors, an ASCII/Binary/hex chart has been included in the appendix of this book. In addition to this, you can find several conversion web pages and programs online, and if all else fails, the Windows calculator will convert hex to binary to decimal to octal, once it has been set to scientific mode.

3.3 The ARM Processor

The Advanced RISC Microprocessor (ARM) is a low-power 32 bit microprocessor based on the Reduced Instruction Set Computer (RISC) principles. In particular, the ARM is used in small devices that have a limited power source and low threshold for heat, such as PDA's, telecommunication devices, and other miniature devices that require a relatively high level of computing power.

3.3.1 Registers

There are a total of 37 registers within this processor that are used to hold values used in the execution of code. Six of these registers are used to hold status values needed to hold the results of compare and mathematical operations, among others. These leaves 31 left to the use of the program, of which a max of 16 are generally available to the programmer. Of these 16, Register 15 (R15) is used to hold the Program Counter (PC), which is used by the processor to keep track of where in the program it is currently executing. R14 is also used by the processor as a subroutine link register (Lr), which is used to temporarily hold the value held by R15 when a Branch and Link (BL) instruction is executed. Finally R13, known as the Stack Pointer (Sp), is used by the processor to hold the memory address of the stack, which is used to store all values about to be used by the processor in it execution.

In addition to these first 16 registers, a debugger allows the programmer to monitor the last four registers (28-31), which are used to hold conditional values. These registers are used to hold the results of arithmetic and logical operations performed by the processor (e.g. addition, subtraction, compares, etc.). The following lists the register and its name/purpose. They are listed in descending order due to the fact that the processor bits are read from high to low.

R31: Negative / Less Tha**n**
R30: Zero
R29: Carry / Borrow / Extend
R28: Overflow

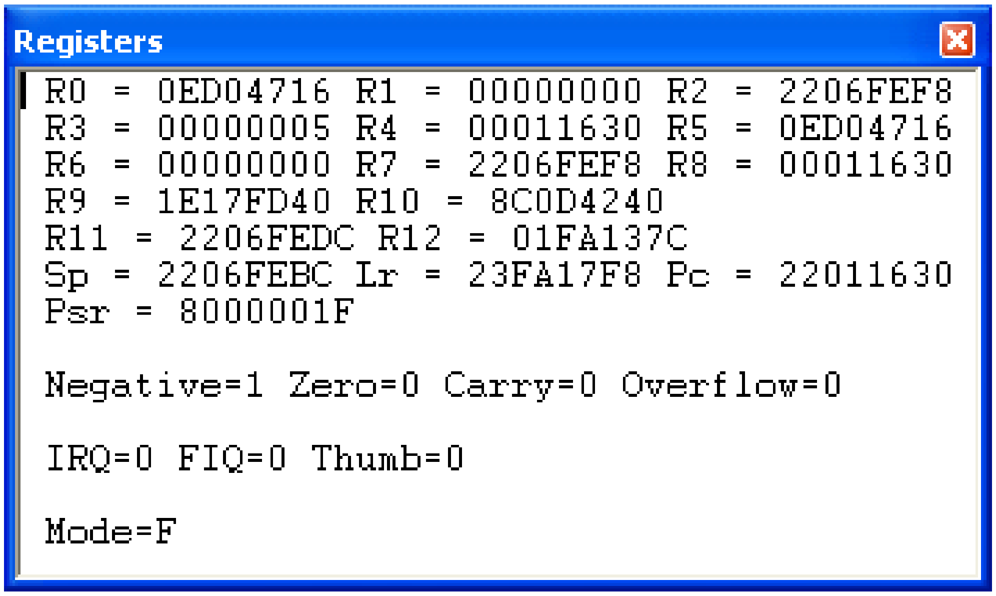
Understanding these registers is very important when debugging software. By knowing what each of these values means, you can be sure to know the next step the program will make. In addition, using a good debugger, you can often alter these values on the fly, thus maintaining 100% control over how a program flows. The following is a table of the possible values and their meanings.

Value	**Meaning**
EQ **NE**	– Z set (equal) – Zero clear (not equal)
CS	– Carry set (unsigned higher or same)
CC	– Carry clear (unsigned lower)
MI	– Negative set
PL	– Negative clear
VS	– Overflow set
VC	– Overflow clear
HI	– Carry set and Zero clear (unsigned hi)
LS **GE** **LT** **GT**	– Carry clear and Zero set (unsigned lo or same) – Negative set and Overflow set or Negative clear and Overflow clear (>=) – Negative set and Overflow clear or Negative clear and Overflow set (<) – Zero clear, and either Negative set and Overflow set, or Negative clear and Overflow clear (>)
LE	– Zero set, and either Negative set and Overflow clear, or Negative clear and

Value	Meaning
	Overflow set (<=)
AL	– Always
NV	– Never

Table 1: ARM Status Codes

Figure 1 illustrates Microsoft's eMbedded Visual Tools (MVT) debugger showing us the values held in registers 0-12, Sp, Lr, and PC. In addition, this figure also let's us see the four registers (R31-R28) used to hold the conditional values. See if you can determine the current status of the program using table 1.



```
R0 = 0ED04716 R1 = 00000000 R2 = 2206FEF8
R3 = 00000005 R4 = 00011630 R5 = 0ED04716
R6 = 00000000 R7 = 2206FEF8 R8 = 00011630
R9 = 1E17FD40 R10 = 8C0D4240
R11 = 2206FEDC R12 = 01FA137C
Sp = 2206FEBC Lr = 23FA17F8 Pc = 22011630
Psr = 8000001F

Negative=1 Zero=0 Carry=0 Overflow=0

IRQ=0 FIQ=0 Thumb=0

Mode=F
```

Figure 1: EVT illustrating the the registers

Now that you understand how the status flags are updated, the following will help you put that knowledge to some practical use. Again, being able to recognize how a program works is essential to reverse-engineering.

Example Scenario #1

CMP #1, #0

In this case you can see that we are comparing two simple values. In real life, the #1 would be represented by a register (e.g. R0, R3, etc.), and #0 would be either a value or another register. To determine how this comparison will alter the status flags, use the following set of questions.

N: If #1 < #0 then N = 1, else N = 0 → N = 0

Z: If #1 = #0 then Z = 1, else Z = 0 → Z = 0
C: If #1 >= #0 then C = 1, else C = 0 → C = 1
O: Was there an error in the calculation, if so, O = 0 → O = 0

Using the above, determine the following.

CMP 23, 36
Negative: If 23 < 36 then N = 0, else N = 0 → N = 1
Zero (Equal): If 23 = 36 then Z = 1, else Z = 0 → Z = 0
Carry: If 23 >= 36 then C = 1, else C = 0 → C = 0
Overflow: Was there an error in the calculation, if so, O = 0 → O = 0

Now that you see how this work in the case of a CMP, we need to look at how the status flags are updated in other situations. The next will illustrate how the flags are updates in the case of a MOVS opcode and an ANDS opcode.

MOVS R1, R0
In this case, you need to look at the status flags as they are labeled and update them according to the value of R0. Use the following steps to determine the outcome.

N: If R0 < 0 then N = 1, else N = 0
Z: if R0 = 0 then Z = 1, else Z = 0

Two things to note from this example, the first is that R0 has to be a negative number for the N flag to be set. This is possible, but only if the binary value starts with a 1. One common value you will see is 0xFFFFFFFF. The second item to note is that the carry value is not updated using the MOVS opcode.

ANDS R1, R0, 0xff

In the case of an ANDS opcode, the results are similar to that of the MOVS opcode. The R0 value is used to determine the flags' status. Use the following to determine the output of the N and Z flags.

N: If R0 < 0 then N = 1, else N = 0
Z: if R0 = 0 then Z = 1, else Z = 0

There are many other opcodes that update the status flags. Some opcodes are implicit and do not require the specification of the 'S'. These update the status flags similar to the CMP opcode. The opcodes that have an explicit 'S' operate like the MOVS example.

3.3.2 ARM Opcodes

The ARM processor has a pre-defined set of operation codes (opcodes) that allows a programmer to write code. These same opcodes are used by compilers, such as Microsoft's EVC, when a program is created for an ARM device. In addition to creating programs, the opcodes are also used when a program is disassembled and/or debugged. For this reason, it is important that you have a understanding of how opcodes are used, and be able to recognize at least the most common opcodes, as well as what operation they perform. The more you are familiar with the opcodes, the easier it will be to determine what the code is doing. In addition, it is also important for you to have some reference of the hex equivalent of an opcode. You will need this to find and replace an opcode as it appears in a hex dump of the file. While practice will ingrain the popular opcodes into your memory, this short discussion will help get you started.

3.3.2.1 Branch (B)

The Branch opcode tells the processor to jump to another part of program, or more specifically the memory, where it will continue its execution. The B opcode is not to be confused with the Branch with Link (BL) opcode discussed next. The main difference is found in the fact that the B opcode simply is a code execution redirector. The program will jump to the specified address and continue processing the instructions. The BL opcode also redirects to another piece of code, but it will eventually jump back to the original code and continue executing where it left off.

There are several variations of the B opcode, most of which make obvious sense. The following is a list of the three most common variants and what they mean. Note that this list relates to the condition table in the previous section. In addition, we have also included the hex code that you will need to search for when altering a branch operation. This is only a partial list. For a full list please visit the references section at the end of this paper.

B	Branch	Always branches	XX XX XX EA
BEQ	B if equal	B if Z flag = 0	XX XX XX 0A
BNE	B if no equal	B if Z flag = 1	XX XX XX 1A

Examples:

B	loc_11498	07 00 00 EA
BEQ	loc_1147C	0C 00 00 0A
BNE	loc_11474	06 00 00 1A

3.3.2.2 Branch with Link (BL)

When a program is executing, there are situations where the program must branch out and process a related piece of information before it can continue with the main program, such as system calls (i.e. a message box). This is made possible with a Branch with Link opcode. Unlike its relative, the B opcode, BL always returns back to the code it originally was executing. To facilitate this, register 14 is used to hold the original address from which the BL was executed and the stack is used to hold any important register values.

The BL opcode has several variants to its base instruction, just like the B opcode. The following is a list of the same three variants and what they mean, which will be followed by examples. It is important to note that the examples show function calls instead of address locations. However, if you look at the actual code you will find normal address, just like the B opcode. The function naming is due to the fact that many BL calls are made to defined function that will return a value or perform a service. As you investigate RVEing, you will become very intimate with the BL opcode. Note, the MVT debugger will not jump to the BL address when doing a line by line execution. It will instead perform the function and continue to the next line. If you want to watch the code specified by the BL operate, you will need to specify a breakpoint at the memory address it branches to. This concept will be discussed later in this paper.

BL	Branch with Link	Always branches	XX XX XX EB
BLEQ	BL if = equal	BL if Z flag = 0	XX XX XX 0B
BLNE	BL if not equal	BL if Z flag = 1	XX XX XX 1B

Examples:

BL AYGSHELL_34	7E 00 00 EB
BLEQ mfcce300_699	5E 3E 00 0B

3.3.2.3 Move (MOV)

A program is constantly moving data around. To facilitate this, registers are updated with values from other registers and with hard coded integers. These values will then be used by other operations to make decisions or perform calculations. This is the purpose of the MOV opcode.

MOV does just what its name implies; it moves information. In addition to basic moves, this opcode also has the same conditional variants as the B and BL opcode. However, by this point you should have the general understanding of what the EQ/NE/etc. means to an instruction set, so it will not be discussed further. Note, most every opcode includes some form of a conditional variant.

It is important to understand how the MOV instruction works. This command can move the value of one register into another, or it can move a hard coded value into a register. However, you should note that the item receiving the data is always a register. The following will list several examples of the MOV command, what it will do, and its hex equivalent.

MOV	R2, #1	01 20 A0 E3	Moves the value 1 into register 2
MOV	R3, R1	01 30 A0 E1	Moves value in R1 into R3
MOV	LR, PC	0F E0 A0 E1	Moves value of R15 into R14*
MOV	R1, R1	01 10 A0 E1	Moves value R1 into R1**

* When a call is made to another function, the value of the PC register, which is the current address location, needs to be stored into the Lr (14) register. This is needed, as previously mentioned, to hold the address from which BL instruction will need to return.

** When RVE, you will need ways to create a non-operation. The infamous NOP slide using 0x90 will not work (as explained later). Instead, you will need to use the MOV opcode to move a registers value into itself. Nothing is updated, and no flags are changed when this operation is executed.

3.3.2.4 Compare (CMP)

In a program, a need often arises in which two pieces of information have to be compared. The results of the comparison are used in numerous ways, from validation of a serial number, to continuation of a counting loop. The assembler instruction that is responsible for this is CMP.

The CMP operation can be used to compare the values in two registers with each other, or a register value and a hard coded value. The results of the comparison do not ouput any data, but it does change the status flags. As previously discussed, if the two values are equal, the Zero flag is set to 0, if the values are not equal, the flag is set to 1. This Zero value is then used by a following opcode to control how or what is executed.

The CMP operation is used in almost every serial number validation. This is accomplished in two ways. The first is the actual check of the entered serial number with a hard coded serial number, which can also be done using system functions (i.e. strcmp). The second is used after the validation check when the program is deciding what piece of code is to be executed next. Typically, there will be a BEQ or BNE operation that uses the status of the Zero flag to either send a 'Wrong Serial Number' message to the screen or to accept the entered serial and allow access to the protected program. This use of the CMP operation will be discussed further in the example part of this paper.

Another use of the CMP is in a loop function. These are very common because they are used to assist in counting, string comparisons, file loads, and more. As a result, being able to recognize a loop in a sequence of assembler programming is an important part of successfully reverse engineering. The following will provide you with an example of how a loop looks when debugging a program.

```
00002AEC        ADD     R1, R4, R7
00002AF0        MOV     R0, R6
00002AF4        BL      sub_002EAC
00002AF8        ADD     R5, R5, #20
00002AFC        ADD     R2, R5, #25
00002A00        CMP     R3, R2
00002A04        BEQ     loc_002AEC
```

This is a simple loop included in an encryption scheme. In memory address 2A04 you can see a Branch occurs if the Zero flag is set. This flag is set, or unset, by the CMP operation at memory address 2A00, which compares the values between R3 and R2. If the values match, the code execution will jump back to memory address 2AEC.

The following is an example of two CMP opcodes and their corresponding hex values.

CMP	R2, R3	03 00 52 E1
CMP	R4, #1	01 00 54 E3

3.3.2.5 Load/Store (LDR/STR)

While the registers are able to store small amount of information, the processor must access the space allotted to it in the RAM to store larger chunks of information. This includes screen titles, serial numbers, colors, settings, and more. In fact, most everything that you see when you use a program has at one time resided in memory. The LDR and STR opcodes are used to write and read this information to and from memory.

While related, these two commands do opposite actions. The LDR instruction loads data from memory into a register and the STR instruction is used to store the data from the registry into memory for later usage. However, there is more to this instruction than the simple transfer of data. In addition to defining where the data is moved, the LDR/STR command have variations that tell the processor how much data is to be moved. The follow is a list of these variants and what they mean.

LDR/STR: Move a Words (four bytes) worth of data to or from memory.
LDRB/STRB: Move a Bytes worth of data to or from memory.
LDRH/STRH: Move two Bytes worth of data to or from memory.

LDR/STR commands are different from the other previously discussed instructions in that they almost always have three pieces of information included with them. This is due to the way in which the load and store instructions work. Since only a few bytes of data are moved at best, the program must keep track of where it was last writing to or reading from. It must then append or read from where it left of from the last read/write. For this reason, you will often find LDR/STR commands in a loop where they will read in or write out large amounts of data, one byte at a time.

The LDR/STR instructions are also different from other instructions in that they typically have three variables controlling where and what data is manipulated. The first variable is the data that is actually being transferred. The second and third determine where the data is written, and if it is manipulated before it is permanently stored or loaded. The follow lists several examples of how these instruction set are used.

STR	R1, [R4, R6]	Store R1 in R4+R6
STR	R1, [R4,R6]!	Store R1 in R4+R6 and write the address in R4
STR	R1, [R4], R6	Store R1 at R4 and write back R4+R6 to R4
STR	R1, [R4, R6, LSL#2]	Store R1 in R4+R6*2 (LSL discussed next)
LDR	R1, [R2, #12]	Load R1 with value at R2+12.
LDR	R1, [R2, R4, R6]	Load R1 with R2+R4+R6

While this provides a good example of how the LDR/STR are used, you should have noted two new items that impacted how the opcode performed. The first is the "!" character that is used to tell the instruction to write back the new information into one of the registers. The second is the use of the LSL command, which is discussed following this segment.

Also related to this instruction is the LDM/STM instructions. These are also used to store or load register values, only they do it on a larger scale. Instead of just moving one value, like LDR/STR, the LDM/STM instruction stores or loads ALL the register values. This is most commonly used when a BL occurs. When this happens, the program must be able to keep track of the original register values, which will be overwritten with values used by the BL code. So, the STM opcode is used to store key registers onto the stack memory, and when the branches code is completely executed, the original register values are loaded back into the registers from memory using the LDM opcode. See the following chunk of code for an example.

```
STMFD   SP!, {R4,R5,LR}
MOV     R0, R4  and more code
LDMFD   SP!, {R4,R5,LR}
```

In this example, R4, R5, and the LR values are placed into memory using the stack pointer address, which is then updated with the new memory address to account for the growth of the stack. At then end of the algorithm, R4, R5, and LR are loaded back from the stack pointer, which is again updated, and the program execution continues.

You should be getting slightly confused at this point. If you are not, then you probably have had previous experience with assembler, or are just a borne programmer. Don't be discouraged if you are feeling overwhelmed, for learning how to program in assembler typically takes months of dedicated study. Fortunately, in the case of reverse engineering, you don't have to know how to program, but just need to be able to figure out what a program is doing.

3.3.2.6 Shifting

The final instruction sets we will look at are the shifting operations. These are somewhat complicated, but a fundamental part of understanding assembler. They are used to manipulate data held by a register at the binary level. In short, they shift the bit values left or right (depending on the opcode), which changes the value held by the register. The follow illustrates how this works with the two most common shifting instruction sets; LSL and LSR. For the sake of space, we will only be performing shifts on bits 0-7 of a 32 bit value.

LSL: Logical Shift Left – Shift the binary values *left* by x number of places, using zeros to fill in the empty spots.

48	110000	Rsl #1 (2)	11000	27
48	110000	Rsl #2 (4)	1100	12
48	110000	Rsl #3 (9)	110	6
48	110000	Rsl #4	11	3

3	0011	Lsl #1 (2)	0110	6
3	0011	Lsl #2 (4)	1100	12
3	0011	Lsl #3 (9)	11000	27
3	0011	Lsl #4	110000	48

LSR: Logical Shift Right – Shift the 32 bit values *right* by x number of places, using zeros to fill in the empty spots.

While these are the most common shift instructions, there are three others that you may see. They are Arithmetic Shift Left (ASL), Arithmetic Shift Right (ASR), and Rotate Right Extended (ROR). All of these shift operations perform the same basic function as LSL/LSR, with some variations on how they work. For example, the ASL/ASR shifts fill in the empty bit places with the bit value of register 31. This is used to preserve the sign bit of the value being held in the register. The ROR shift, on the other hand, carries the bit value around from bit 0 to bit 31.

3.4 Summary

The previous pages have given you an inside look at the assembler programming language. You will need this information later in this paper when we practice some of our RVE skills on a test program. This information is invaluable to you as you attempt to debug software, looking for holes and security risks.

4 Reverse-Engineering Tools

Reverse engineering software requires several key tools. Each tool allows its user to interact with the target program in one specific way, and without these tools the reverse engineering process can take much longer. The following is a breakdown of the types of tools, and an example of each.

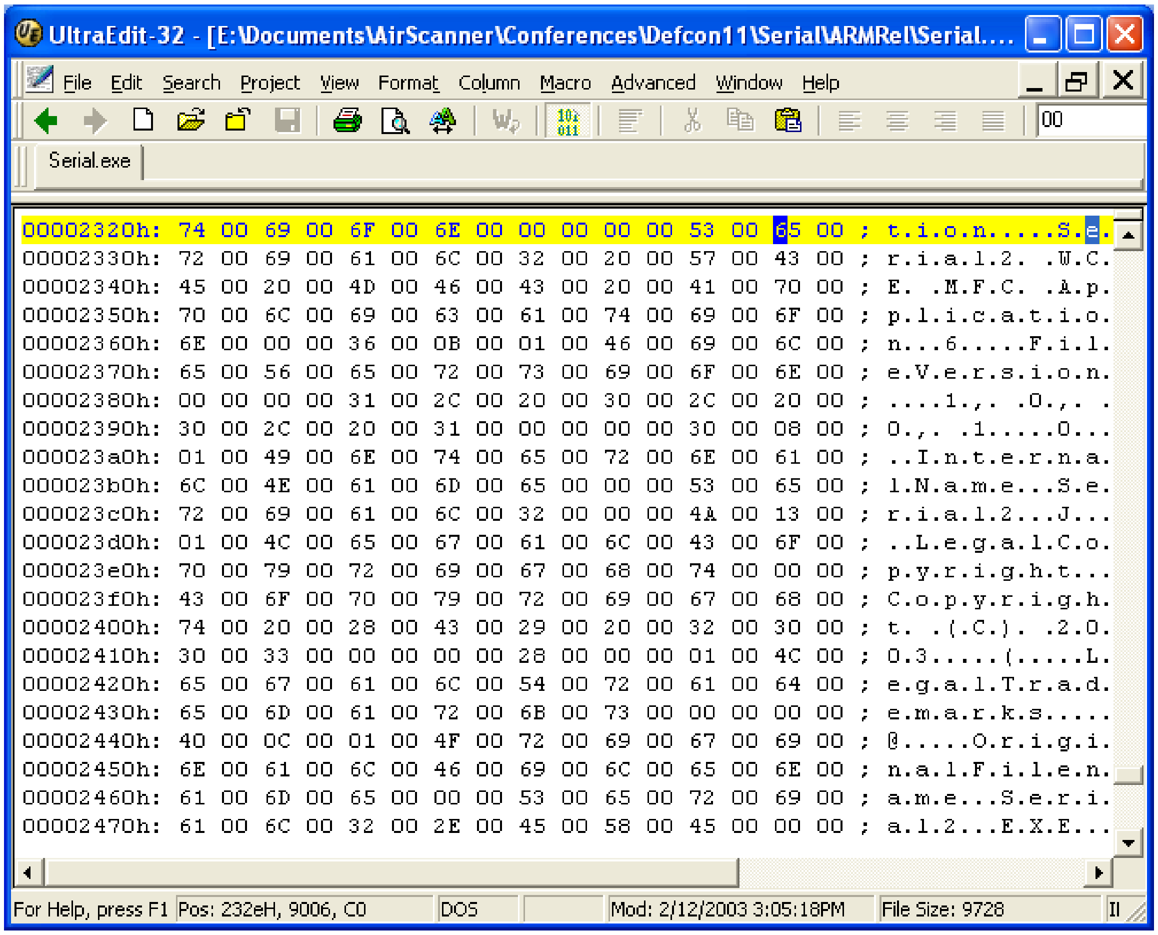
4.1 Hex Editor

As previously described, all computer data is processed as binary code. However, this code is rather difficult to follow for the human eye, which lead to the creation of hex. Using the numbers between 0-9 and the letters A-F, any eight digit binary value can be quickly converted to a one or two character hex value, and vise versa.

While it is importance that a hex editor can convert the program file to its hex equivalent, what is more important is that a hex editor allows a user to easily alter the hex code to new values. In addition to basic manipulation, some hex editors also provide search tools, ASCII views, macros and more.

UltraEdit-32

UltraEdit-32 is a windows based hex editing program that does all of the above and more. As you can see from figure 2, UltraEdit-32 contains the three basic, but very necessary fields required to edit hex code. The memory address on the left is used to locate the particular characters that you want to change. This address will be provided by the disassembler, which is discussed next. Once the correct line is located, the next step is to find the hex code in the line that represents the information you want to alter. The ASCII view, which is not always necessary, can provide some interesting and useful information, especially if you were changing plain text information in a file.



UltraEdit-32 - [E:\Documents\AirScanner\Conferences\Defcon11\Serial\ARMRel\Serial....

```
00002320h: 74 00 69 00 6F 00 6E 00 00 00 00 00 53 00 65 00 ; t.i.o.n.....S.e.
00002330h: 72 00 69 00 61 00 6C 00 32 00 20 00 57 00 43 00 ; r.i.a.l.2. .W.C.
00002340h: 45 00 20 00 4D 00 46 00 43 00 20 00 41 00 70 00 ; E. .M.F.C. .A.p.
00002350h: 70 00 6C 00 69 00 63 00 61 00 74 00 69 00 6F 00 ; p.l.i.c.a.t.i.o.
00002360h: 6E 00 00 00 36 00 0B 00 01 00 46 00 69 00 6C 00 ; n...6.....F.i.l.
00002370h: 65 00 56 00 65 00 72 00 73 00 69 00 6F 00 6E 00 ; e.V.e.r.s.i.o.n.
00002380h: 00 00 00 00 31 00 2C 00 20 00 30 00 2C 00 20 00 ; ....1.,. .0.,. .
00002390h: 30 00 2C 00 20 00 31 00 00 00 00 00 30 00 08 00 ; 0.,. .1.....0...
000023a0h: 01 00 49 00 6E 00 74 00 65 00 72 00 6E 00 61 00 ; ..I.n.t.e.r.n.a.
000023b0h: 6C 00 4E 00 61 00 6D 00 65 00 00 00 53 00 65 00 ; l.N.a.m.e...S.e.
000023c0h: 72 00 69 00 61 00 6C 00 32 00 00 00 4A 00 13 00 ; r.i.a.l.2...J...
000023d0h: 01 00 4C 00 65 00 67 00 61 00 6C 00 43 00 6F 00 ; ..L.e.g.a.l.C.o.
000023e0h: 70 00 79 00 72 00 69 00 67 00 68 00 74 00 00 00 ; p.y.r.i.g.h.t...
000023f0h: 43 00 6F 00 70 00 79 00 72 00 69 00 67 00 68 00 ; C.o.p.y.r.i.g.h.
00002400h: 74 00 20 00 28 00 43 00 29 00 20 00 32 00 30 00 ; t. .(.C.). .2.0.
00002410h: 30 00 33 00 00 00 00 00 28 00 00 00 01 00 4C 00 ; 0.3.....(.....L.
00002420h: 65 00 67 00 61 00 6C 00 54 00 72 00 61 00 64 00 ; e.g.a.l.T.r.a.d.
00002430h: 65 00 6D 00 61 00 72 00 6B 00 73 00 00 00 00 00 ; e.m.a.r.k.s.....
00002440h: 40 00 0C 00 01 00 4F 00 72 00 69 00 67 00 69 00 ; @.....O.r.i.g.i.
00002450h: 6E 00 61 00 6C 00 46 00 69 00 6C 00 65 00 6E 00 ; n.a.l.F.i.l.e.n.
00002460h: 61 00 6D 00 65 00 00 00 53 00 65 00 72 00 69 00 ; a.m.e...S.e.r.i.
00002470h: 61 00 6C 00 32 00 2E 00 45 00 58 00 45 00 00 00 ; a.l.2...E.X.E...
```

For Help, press F1 | Pos: 232eH, 9006, C0 | DOS | Mod: 2/12/2003 3:05:18PM | File Size: 9728 | II

Figure 2 Ultra Edit screen shot

4.2 Disassemblers

While it would be possible for a person to reverse-engineering a program, as it exists in hex format, it would be very difficult and would require a very deep understanding of hex and binary code. Since this level of knowledge is impractical, the concept of the disassembler was designed to help us humans find a workable method of communicating with a computer.

As we previously discussed, there are several levels of languages. The upper languages like Visual Basic are easy to follow and program with. This is because the syntax of the language follows spoken language. Unfortunately, a computer cannot directly understand upper level languages. So, after a program is written, it is compiled, or rewritten using code a computer can understand. This code, while it actually exists as hex or binary, can easily be converted to a low level language known as assembler.

Assembler code is relatively basic, once you understand it. While the syntax is different for each processor type (e.g. RISC, Intel, Motorola), the general commands are relatively the same. Using a set of opcodes, assembler controls the processor and how it interacts with RAM and other parts of the computer. In other words, assembler speaks to the heart and nervous systems of the computer.

Once a program is compiled, it creates a hex file (or set of files) that the computer loads into memory and reads. As previously mentioned, this code is stored as hex, but is converted to its binary equivalent and then process by the computer. To facilitate human understanding, a disassembler takes the same hex file and converts it to assembler code. Since there is no real difference, other than format and appearance, a person can use the assembler code to see the path the program takes when it is executed. In addition, a good disassembler will also provide the user with the information they need to alter the assembler code, through the use of a hex editor. By researching the code, a hacker can find the point in the program, for example, that a serial number is checked. They could then look up the memory location, and use the hex editor to remove the serial number check from the program.

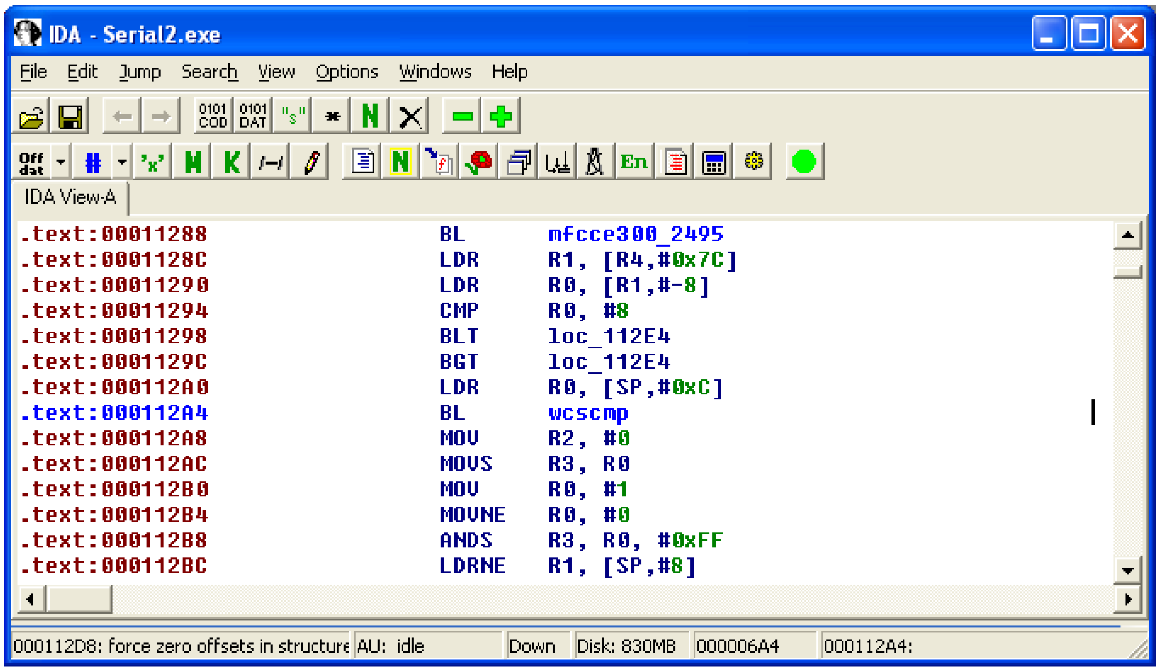
IDA Pro

By far, IDA Pro (The Interactive Disassembler) is one of the best disassembler programs on the market. In fact, "**IDA Pro was a 2001 PC Magazine Technical Excellence Award Finalist,** defeated only by Microsoft and its Visual Studio .NET", according to their web site. While there are many other disassemblers available, some for free, this program does a marvelous job providing its user with a wide selection of supported processor types, and a plethora of tools to assist in disassembling.

While we could spend an entire paper delving into the many features and functionality of this program, it is outside the scope of this paper. However, there are several key features that need to be outlined.

The first feature is that this program supports more processors than any other disassembler, and the list keeps growing. In other words, you can disassemble everything from the processor used in an iPAQ to your desktop computer. In addition to this, IDA includes several proprietary features that help to identify many of the standard function calls made in a program. By providing this information, you do not have to track the code to the function yourself. Another proprietary feature includes the incorporation of a C like programming language that can be used to automate certain routines, such as decryption.

However, it is the ease of use and amount of information provided by IDA Pro that makes it a great program. As you can see in figure 3 IDA Pro provides a wealth of information, much of it I can't fit on one screen shot. However, in this one shot, you can IDA has located the functions that are being called, and used their names in place of the memory address that would normally exist. In addition, you can see that IDA has listed all the window names, and provides a colorful look at how the data is laid out in the memory. In IDA 4.21+, the program also provides a graphical diagram of how functions are tied together.



```
.text:00011288                 BL      mfcce300_2495
.text:0001128C                 LDR     R1, [R4,#0x7C]
.text:00011290                 LDR     R0, [R1,#-8]
.text:00011294                 CMP     R0, #8
.text:00011298                 BLT     loc_112E4
.text:0001129C                 BGT     loc_112E4
.text:000112A0                 LDR     R0, [SP,#0xC]
.text:000112A4                 BL      wcscmp
.text:000112A8                 MOV     R2, #0
.text:000112AC                 MOVS    R3, R0
.text:000112B0                 MOV     R0, #1
.text:000112B4                 MOVNE   R0, #0
.text:000112B8                 ANDS    R3, R0, #0xFF
.text:000112BC                 LDRNE   R1, [SP,#8]
```

```
000112D8: force zero offsets in structure AU: idle  Down  Disk: 830MB  000006A4  000112A4:
```

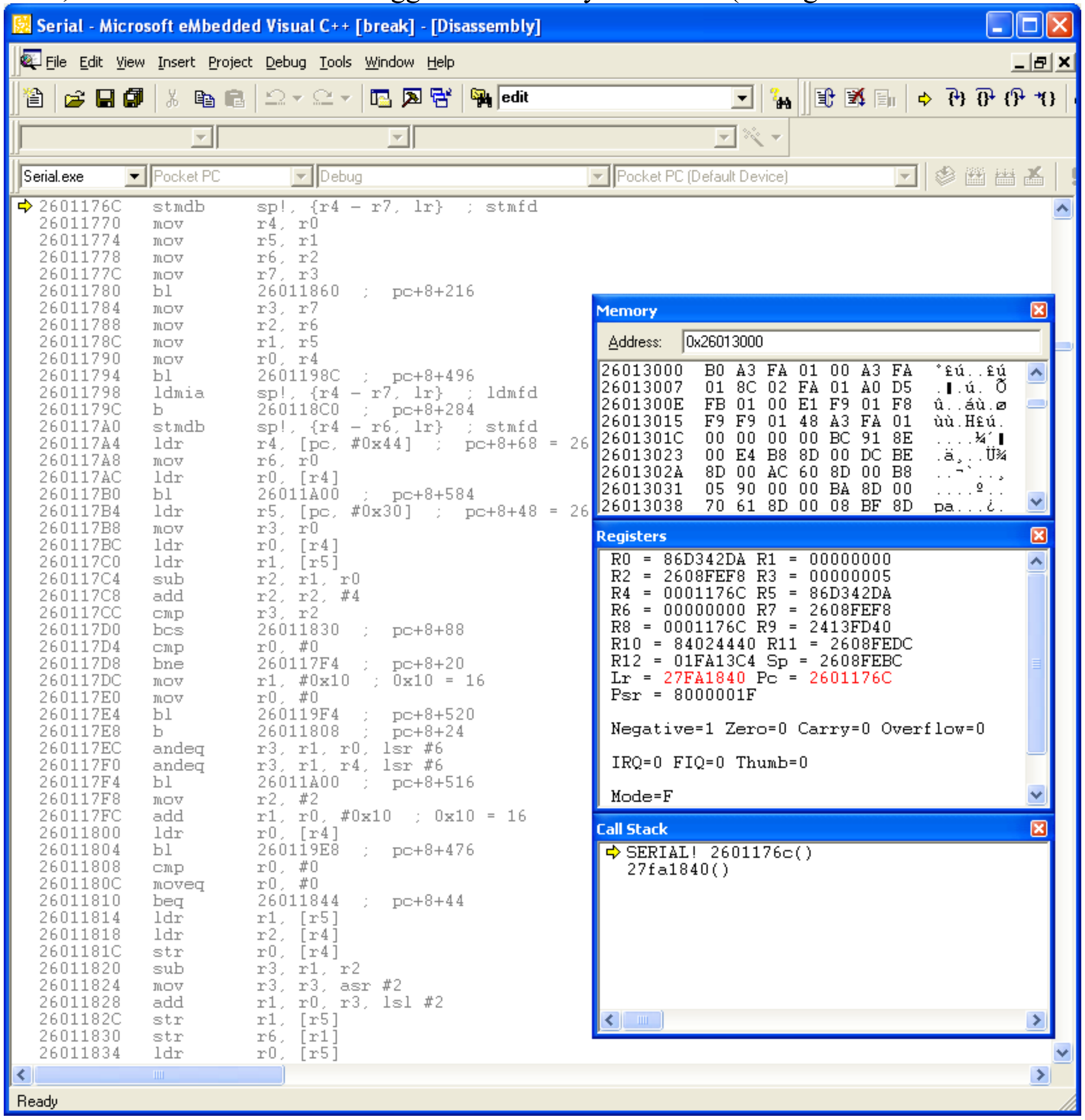
Figure 3 IDA Pro Screen shot.

4.3 Debuggers

The debugger is the third major program in any reverse-engineers bag of tools. This program is used to watch the code execute live and helps the reverse engineer watch the action in real time. Using such a tool, a reverse-engineer can monitor the register values, status flags, modes, etc. of a program as it executes. In addition, a good debugger will allow the user to alter the values being used by the program. Memory, registers, status flags and more can be updated to control the direction of the programs execution. This type of control can reduce the amount of time a user has to spend working with raw assembler code.

Due to the limited environment of the PPC platform, and in particular the Pocket PC OS, there are few choices for a debugger. In fact, there is really only one that actually debugs live on the PPC device. This debugger is free and is actually included with the SDK from

Microsoft. To obtain it, go to www.microsoft.com and download embedded Visual Tools. This will come with VB and VC++ for the Pocket PC OS. While these are programming tools, the VC++ includes a debugger that is ready for action (see figure 4.


Serial - Microsoft eMbedded Visual C++ [break] - [Disassembly]
File Edit View Insert Project Debug Tools Window Help
edit
Serial.exe
Pocket PC
Debug
Pocket PC (Default Device)
Memory
Registers
Call Stack
Ready

```
2601176C  stmdb   sp!, {r4 - r7, lr}  ; stmfd
26011770  mov     r4, r0
26011774  mov     r5, r1
26011778  mov     r6, r2
2601177C  mov     r7, r3
26011780  bl      26011860  ;  pc+8+216
26011784  mov     r3, r7
26011788  mov     r2, r6
2601178C  mov     r1, r5
26011790  mov     r0, r4
26011794  bl      2601198C  ;  pc+8+496
26011798  ldmia   sp!, {r4 - r7, lr}  ; ldmfd
2601179C  b       260118C0  ;  pc+8+284
260117A0  stmdb   sp!, {r4 - r6, lr}  ; stmfd
260117A4  ldr     r4, [pc, #0x44]  ;  pc+8+68 = 26
260117A8  mov     r6, r0
260117AC  ldr     r0, [r4]
260117B0  bl      26011A00  ;  pc+8+584
260117B4  ldr     r5, [pc, #0x30]  ;  pc+8+48 = 26
260117B8  mov     r3, r0
260117BC  ldr     r0, [r4]
260117C0  ldr     r1, [r5]
260117C4  sub     r2, r1, r0
260117C8  add     r2, r2, #4
260117CC  cmp     r3, r2
260117D0  bcs     26011830  ;  pc+8+88
260117D4  cmp     r0, #0
260117D8  bne     260117F4  ;  pc+8+20
260117DC  mov     r1, #0x10  ; 0x10 = 16
260117E0  mov     r0, #0
260117E4  bl      260119F4  ;  pc+8+520
260117E8  b       26011808  ;  pc+8+24
260117EC  andeq   r3, r1, r0, lsr #6
260117F0  andeq   r3, r1, r4, lsr #6
260117F4  bl      26011A00  ;  pc+8+516
260117F8  mov     r2, #2
260117FC  add     r1, r0, #0x10  ; 0x10 = 16
26011800  ldr     r0, [r4]
26011804  bl      260119E8  ;  pc+8+476
26011808  cmp     r0, #0
2601180C  moveq   r0, #0
26011810  beq     26011844  ;  pc+8+44
26011814  ldr     r1, [r5]
26011818  ldr     r2, [r4]
2601181C  str     r0, [r4]
26011820  sub     r3, r1, r2
26011824  mov     r3, r3, asr #2
26011828  add     r1, r0, r3, lsl #2
2601182C  str     r1, [r5]
26011830  str     r6, [r1]
26011834  ldr     r0, [r5]
```

Memory — Address: 0x26013000

```
26013000  B0 A3 FA 01 00 A3 FA  °£ú..£ú
26013007  01 8C 02 FA 01 A0 D5  .▌.ú. Õ
2601300E  FB 01 00 E1 F9 01 F8  û..áù.ø
26013015  F9 F9 01 48 A3 FA 01  ùù.H£ú.
2601301C  00 00 00 00 BC 91 8E  ....¼'▌
26013023  00 E4 B8 8D 00 DC BE  .ä¸..Ü¾
2601302A  8D 00 AC 60 8D 00 B8  ..¬`..¸
26013031  05 90 00 00 BA 8D 00  ....º..
26013038  70 61 8D 00 08 BF 8D  pa...¿.
```

Registers

```
R0 = 86D342DA R1 = 00000000
R2 = 2608FEF8 R3 = 00000005
R4 = 0001176C R5 = 86D342DA
R6 = 00000000 R7 = 2608FEF8
R8 = 0001176C R9 = 2413FD40
R10 = 84024440 R11 = 2608FEDC
R12 = 01FA13C4 Sp = 2608FEBC
Lr = 27FA1840 Pc = 2601176C
Psr = 8000001F

Negative=1 Zero=0 Carry=0 Overflow=0

IRQ=0 FIQ=0 Thumb=0

Mode=F
```

Call Stack

```
SERIAL! 2601176c()
27fa1840()
```

Figure 4:EVC Debugger

We will be demonstrating how this program works in the next section, but the follow is a list of pointers to help you get started.

To get connected to a program on the PPC, copy it first to the PC and open it from the local computer.

- The debugger will copy the file BACK over to the PPC device, so be sure to define where you want the file to go (Project → Settings).
- It is best to launch the program using the F11 key. This is the 'Step into' function and will only load the program and place the pointer at the first line of actual

code. If you use the Run command, you will execute the program as it would normally execute, which could make it difficult to break cleanly at the point you want to debug.

- Make extensive use of breakpoints (Alt-F9)
- Use your disassembler to determine the relative address, which corresponds to the actual address in the debugger.
- If all else fails, use the break option at the top. However, note that this will force a complete reload of your program for further debugging.

In summary, a debugger is not necessary, but is so helpful that it really should be used. Debugging on the PPC platform is painfully slow if you use the standard USB/serial connection that provides the default HotSync connection. If you want much faster access and response time, take the time to configure you PPC device to Sync up over a network connection.

5. Practical Reverse-Engineering

Reverse engineering is not a subject that can be learned by simple reading. In order to understand the intricacies involved, you must practice. This segment will provide a legal and hands on tutorial on how to bypass a serial protection. This will only describe one method of circumvention, of only one protection scheme, which means there is more than one 'right' way to do it. We will use information previously discussed as a foundation.

5.1 Overview

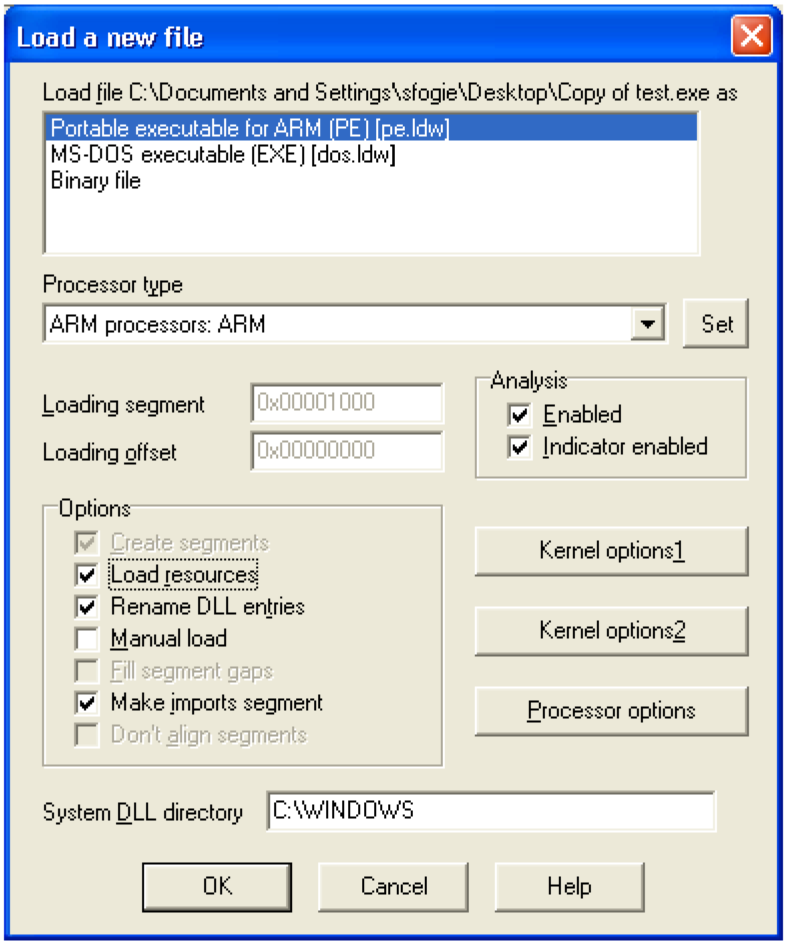
For our example, we will use our own program. This program was written in Visual C++ to provide you with a real working product to test and practice your newly acquired knowledge. Our program simulates a simple serial number check that imitates those of many professional programs. You will see first hand how a cracker can reverse-engineer a program to allow any serial number, regardless of length or value.

5.2 Loading the target

The first step in reverse-engineering a program requires you to tear it apart. This is accomplished via a disassembler program, one of which is IDA Pro. There are many other disassemblers available; however, IDA Pro has earned the respect of both legitimate debuggers and crackers alike for its simplicity, power, and versatility.

To load the target into the disassembler, step through the following steps.

1. Open IDA (click OK through splash screen)
2. Click **[New]** button at Welcome screen and select *test.exe* from hard drive, then click [**Open]**
3. Check the '**Load resources**' box, change the Processor type drop down menu to **ARM processors: ARM** and click **[OK]** as illustrated by figure ??.
4. Click **[OK]** again if prompted to change processor type.


Load a new file

Load file C:\Documents and Settings\sfogie\Desktop\Copy of test.exe as

- Portable executable for ARM (PE) [pe.ldw]
- MS-DOS executable (EXE) [dos.ldw]
- Binary file

Processor type: ARM processors: ARM

Loading segment: 0x00001000

Loading offset: 0x00000000

Analysis
- [✓] Enabled
- [✓] Indicator enabled

Options
- [✓] Create segments
- [✓] Load resources
- [✓] Rename DLL entries
- [ ] Manual load
- [ ] Fill segment gaps
- [✓] Make imports segment
- [ ] Don't align segments

System DLL directory: C:\WINDOWS

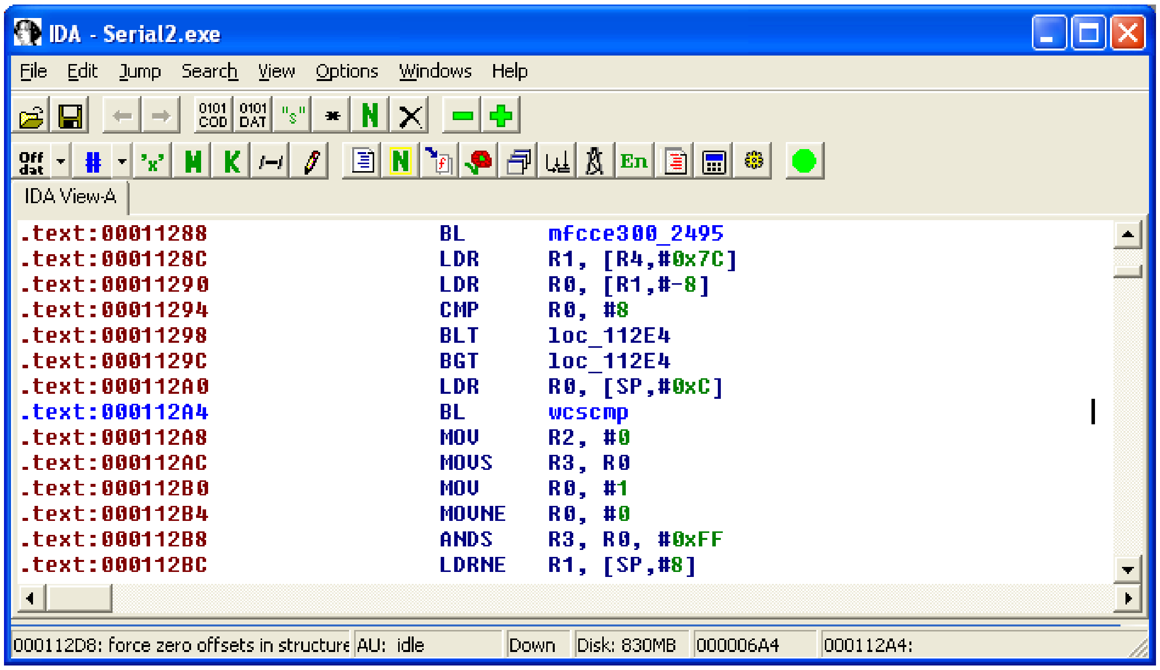
Figure 5 IDA Pro loading screen

4. Locate any requested *.dll file and wait for IDA to disassemble program.

Note: You will need to find the requested files on the Windows CE device and transfer them over to a local folder on your PC. This will allow IDA to fully disassemble the program. The disassembly of serial.exe will require the mfcee300.dll and olece300.dll. Other programs may require different *.dll files, which can be found online or on the PPC device.

5.3 Practical Disassembly

Once the program is open, you should see a screen similar to figure 6, this screen is the default disassembly screen that shows you the program fully disassembled. On the left side of the window, you will see a ".text: ########" column that represents the memory address of the line. The right side of the window holds the disassembled code, which is processed by the PPC device.



```
.text:00011288                 BL      mfcce300_2495
.text:0001128C                 LDR     R1, [R4,#0x7C]
.text:00011290                 LDR     R0, [R1,#-8]
.text:00011294                 CMP     R0, #8
.text:00011298                 BLT     loc_112E4
.text:0001129C                 BGT     loc_112E4
.text:000112A0                 LDR     R0, [SP,#0xC]
.text:000112A4                 BL      wcscmp
.text:000112A8                 MOV     R2, #0
.text:000112AC                 MOVS    R3, R0
.text:000112B0                 MOV     R0, #1
.text:000112B4                 MOVNE   R0, #0
.text:000112B8                 ANDS    R3, R0, #0xFF
.text:000112BC                 LDRNE   R1, [SP,#8]
```

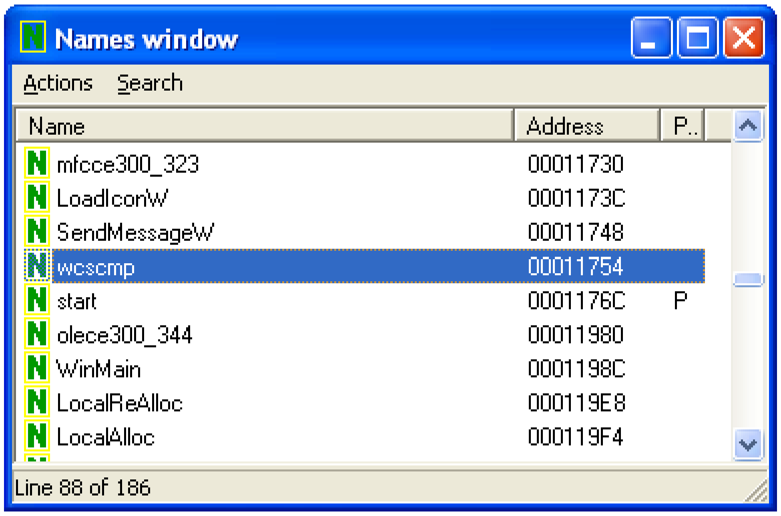
```
000112D8: force zero offsets in structure  AU:  idle  Down  Disk: 830MB  000006A4  000112A4:
```

Figure 6: IDA Pro

In addition to the data on the default screen, you have access to several other important pieces of information, one of the most helpful of which is the *Names window.* This dialog window provides you with a listing of all the functions used by the program. In many ways, this is a bookmark list that can be used to jump to a particular function or call that could lead to a valuable piece of code. In this window you will find names such as, LoadStringW, MessageboxW, wcscmp, wcslen, and more. These are flares to reverse-engineers because they are often used to read in serial numbers, popup a warning, compare two strings, or check the length to be sure it is correct. In fact, some programs call their functions by VERY obvious names, such as ValidSerial or SerialCheck. These programs might as well include a crack with the program for all the security they have. However, it is also possible to throw a cracker off track by using this knowledge to misname windows. Imagine if a program threw in a bogus serial check that only resulted in a popup window that congratulated the cracker of their fine job!

5.4 Locating a Weakness

From here, a cracker basically has to start digging. While our serial.exe is basic, we can still see that the Names window still offers us a place to start. If you scroll through the many names, you will eventually come to the wcscmp function, as illustrated in figure 7. If you recall, this function is used to compare two values together. To access the point in the program where the wcscmp function is located, double click on the line.

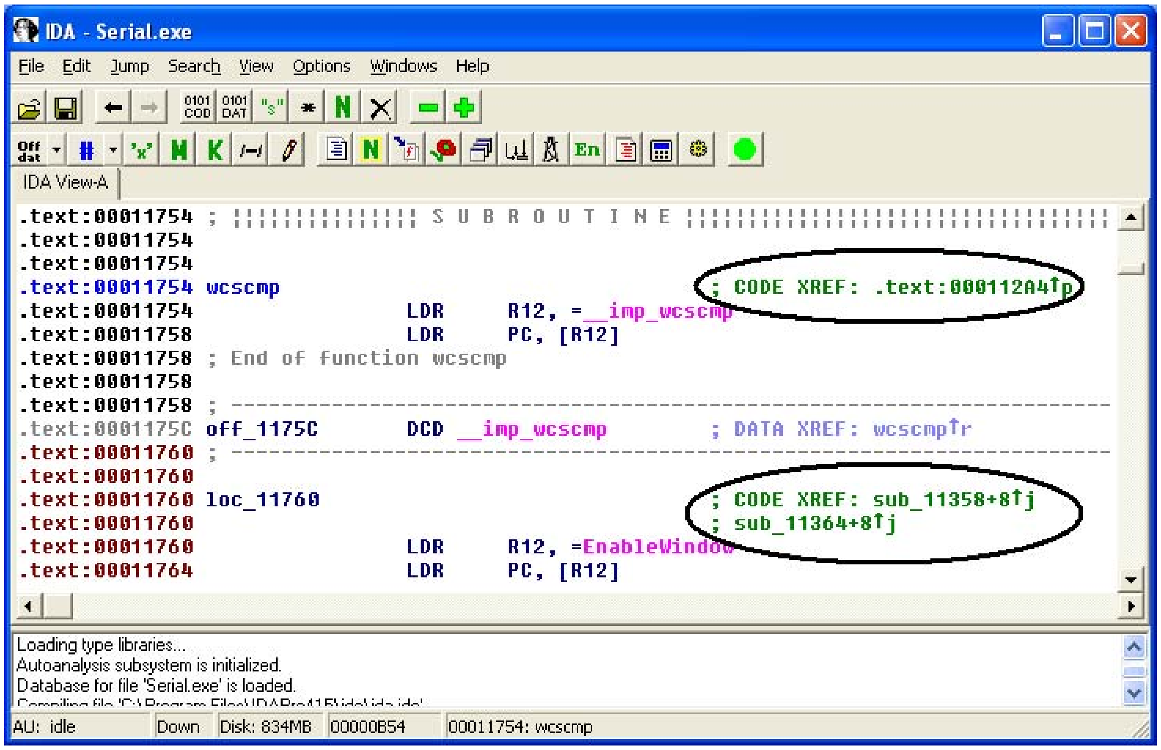


Name	Address	P..
mfcce300_323	00011730	
LoadIconW	0001173C	
SendMessageW	00011748	
wcscmp	00011754	
start	0001176C	P
olece300_344	00011980	
WinMain	0001198C	
LocalReAlloc	000119E8	
LocalAlloc	000119F4	

Line 88 of 186

Figure 7: Locating wcscmp in Names window

Once the IDA disassembly screen moves to the wcscmp function, you should take note of a few items. The first is that this function is an imported function from a supporting .dll (coredll.dll in this case), the second item to notice is the part of the data that is circle in figure 8. At each function, you will find one or two items listed at the top right corner. These list the addresses in the program that the function is used. Note, if there are three dots to the right of the second address listing, this means the function is used more than twice. To access the list of addresses, simply click on the dots. In larger programs, a wcscmp function can be called tens, if not hundreds of times. However, we are in luck, the wcscmp function in serial.exe is only referenced once. To jump to that part of the disassembled program, double click on the address. Once the IDA screen refreshes itself with the location of the selected address, it is time to start rebuilding the program.



```
.text:00011754 ; ||||||||||||||| S U B R O U T I N E |||||||||||||||||||||||||||||||
.text:00011754
.text:00011754
.text:00011754 wcscmp                                   ; CODE XREF: .text:000112A4↑p
.text:00011754                 LDR     R12, =__imp_wcscmp
.text:00011758                 LDR     PC, [R12]
.text:00011758 ; End of function wcscmp
.text:00011758
.text:00011758 ; ---------------------------------------------------------------------------
.text:0001175C off_1175C       DCD __imp_wcscmp         ; DATA XREF: wcscmp↑r
.text:00011760 ; ---------------------------------------------------------------------------
.text:00011760
.text:00011760 loc_11760                                ; CODE XREF: sub_11358+8↑j
.text:00011760                                          ; sub_11364+8↑j
.text:00011760                 LDR     R12, =EnableWindow
.text:00011764                 LDR     PC, [R12]
```

```
Loading type libraries...
Autoanalysis subsystem is initialized.
Database for file 'Serial.exe' is loaded.
```

```
AU: idle    Down    Disk: 834MB    00000B54    00011754: wcscmp
```

Figure 8: Viewing wcscmp function call in IDA

5.5 Reverse-Engineering the Algorithm

Since serial.exe is a relatively simple program, all the code we will need to review and play with is located within a few lines. They are as follows:

```
.text:00011224          MOV    R4, R0
.text:00011228          ADD    R0, SP, #0xC
.text:0001122C          BL     CString::CString(void)
.text:00011230          ADD    R0, SP, #8
.text:00011234          BL     CString::CString(void)
.text:00011238          ADD    R0, SP, #4
.text:0001123C          BL     CString::CString(void)
.text:00011240          ADD    R0, SP, #0x10
.text:00011244          BL     CString::CString(void)
.text:00011248          ADD    R0, SP, #0
.text:0001124C          BL     CString::CString(void)
.text:00011250          LDR    R1, =unk_131A4
.text:00011254          ADD    R0, SP, #0xC
.text:00011258          BL     CString::operator=(ushort)
.text:0001125C          LDR    R1, =unk_131B0
.text:00011260          ADD    R0, SP, #8
.text:00011264          BL     CString::operator=(ushort)
.text:00011268          LDR    R1, =unk_131E0
.text:0001126C          ADD    R0, SP, #4
```

```
.text:00011270              BL      ; CString::operator=(ushort)
.text:00011274              LDR     R1, =unk_1321C
.text:00011278              ADD     R0, SP, #0
.text:0001127C              BL      CString::operator=(ushort)
.text:00011280              MOV     R1, #1
.text:00011284              MOV     R0, R4
.text:00011288              BL      CWnd::UpdateData(int)
.text:0001128C              LDR     R1, [R4,#0x7C]
.text:00011290              LDR     R0, [R1,#-8]
.text:00011294              CMP     R0, #8
.text:00011298              BLT     loc_112E4
.text:0001129C              BGT     loc_112E4
.text:000112A0              LDR     R0, [SP,#0xC]
.text:000112A4              BL      wcscmp
.text:000112A8              MOV     R2, #0
.text:000112AC              MOVS    R3, R0
.text:000112B0              MOV     R0, #1
.text:000112B4              MOVNE   R0, #0
.text:000112B8              ANDS    R3, R0, #0xFF
.text:000112BC              LDRNE   R1, [SP,#8]
.text:000112C0              MOV     R0, R4
.text:000112C4              MOV     R3, #0
.text:000112C8              BNE     loc_112F4
.text:000112CC              LDR     R1, [SP,#4]
.text:000112D0              B       loc_112F4
.text:000112E4
.text:000112E4 loc_112E4                        ; CODE XREF: .text:00011298
.text:000112E4                                  ; .text:0001129C
.text:000112E4              LDR     R1, [SP]
.text:000112E8              MOV     R3, #0
.text:000112EC              MOV     R2, #0
.text:000112F0              MOV     R0, R4
.text:000112F4
.text:000112F4 loc_112F4                        ; CODE XREF: .text:000112C8
.text:000112F4                                  ; .text:000112D0
.text:000112F4              BL      CWnd__MessageBoxW
```

If have not touched anything after IDA placed you at address 0x000112A4, then that line should be highlighted blue. If you want to go back to the last address, use the back arrow at the top of the window or hit the ESC key.

Since we want to show you several tricks crackers will use when extracting or bypassing protection, lets start by considering what we are viewing. At first glance at the top of our code, you can see there is a pattern. A string value appears to be loaded in from program data, and then a function is called that does something with that value. If we double click on unk_131A4, we can see what the first value is “12345678”, or our serial number. While our serial.exe example is simplified, the fact remains that any data used in a programs validation must be loaded in from the actual program data and stored in RAM. As our example illustrates, it doesn't take much to discover to discover a plain text serial number. In addition, it should be noted that any hex editor can also be used to find this value, though it may be difficult to parse out a serial number from the many other character strings that are revealed in a hex editor.

As a result of this plain text problem, many programmers build an algorithm into the program that deciphers the serial number as it is read in from memory. This will typically be indicated by a BL to the memory address in the program that handles the encryption/algorithm. An example of another method of protection is to use the devices owners name or some other value to dynamically build a serial number. This completely avoids the problems surrounding storing it within the program file, and indirectly adds an extra layer of protection on to the program. Despite efforts to create complex and advanced serial number creation schemes, the simple switch of a 1 to a 0 can nullify many anti-piracy algorithms, as you will see.

The remaining code from 0x00011250 to 0x0001127C is also used to load in values from program data to the devices RAM. If you check the values at the address references, you can quickly see that there are three messages that are loaded into memory as well. One is a 'Correct serial' message, and the other two are 'Incorrect serial' messages. Knowing that there are two different messages is a minor, but important tidbit of information because it tells us that failure occurs in stages or as a result of two different checks.

Moving on through the code, we can see that R1 is loaded with some value out of memory, which is then used to load another value into R0. After this, in address 0x00011294, we can see that R0 is compared to the number eight (CMP R0,8). The next two lines then check the results of the comparison, and if it is greater than or less than eight the program jumps to loc_112E4 and continues from there.

If we follow loc_112E4 in IDA Pro, it starts to get a bit more difficult to determine what is happening. This brings us to the second phase of the reverse-engineering process; the live debugger.

5.6 Practical Debugging

Currently, the best debugger for the Pocket PC operating system is Microsoft's eMbedded Visual C++ program (MVC). This program is part of the Microsoft Visual Tools package, which is currently free of charge. Once you download it from Microsoft, or a mirror, install it and open eMbedded Visual C++ (MVC). For the rest of our example, we will be using the serial.exe program currently being dissected by IDA Pro. You will need to have your pocket PC device plugged in and connected to your PC to do live debugging. This can be accomplished using the traditional USB/serial connection, which is very slow, or using a network (wired or wireless) based sync connection that is 100x faster. Use the following instructions to get serial.exe loaded into MVC

1.Obtain a working connection between the PPC and the computer running MVC
2.Start up the **MVC**
3.Click the **Open** folder
4.Switch **Files of type:** to **Executable Files (.exe; .dll;.ocx)**
5.Locate **serial.exe** and click **Open**

Note: Depending on the program you are loading, you may need to adjust the download directory under Project -> Settings -> Debug tab -> Download Directory. This tells the MVC to send a copy of the file to that directory on the Pocket PC, which may be necessary if the program has its own .dlls. Since serial.exe is a one file program, this setting doesn't matter.

6.Hit the **F11** key to execute serial.exe on the Pocket PC and load up the disassembly information.
7.You will see a connecting dialog box, which should be followed by a CPU mismatch warning. Click **Yes** on this warning and the next one. This warning is due to the fact that you loaded MVC with a local copy of serial.exe, and the CPUfor your local system doesn't match the Pocket PC device.
8.Click **OK** for the '...does not contain debugging information.' alert
9.Click **Cancel** on the .dll requests. For serial.exe you will not need these two dll files. However, this is not always the case.

You should now be staring at a screen that looks remarkable similar to IDA Pro. The first thing you will want to do is set a breakpoint at the address location in serial.exe that corresponds to the location of the previously discussed segment of code (e.g. 0X00011280). However, you should take a moment and look at the address column in the MVC. As you will quickly see, IDA Pro's memory addresses and the MVC's do not exactly match.

5.7 Setting Breakpoints

This is because IDA provides a relative address, meaning it will always start at 0. In the MVC, you will be working with an absolute address, which is based on actual memory location, not the allocated memory address as in IDA. However, with the exception of the first two numbers, the addresses will be same. Therefore, take note of the address block that serial.exe has loaded, and set the breakpoint with this value in mind. For example, if the first address in the MVC is 0x2601176C, and the address you found in IDA is 0x00011280, the breakpoint should be set at 0x26011280, which exactly we need to do in our example.

Setting a breakpoint is simple. Simply click Edit → Breakpoints or hit Alt-F9. In the breakpoint window set the breakpoint at '0x26011280', with any changes as defined by the absolute memory block. Once the breakpoint is entered, hit the F5 key to execute the program. You should now see a serial screen on your Pocket PC similar to figure 9. Enter any value in the pocket PC and hit the Submit button.

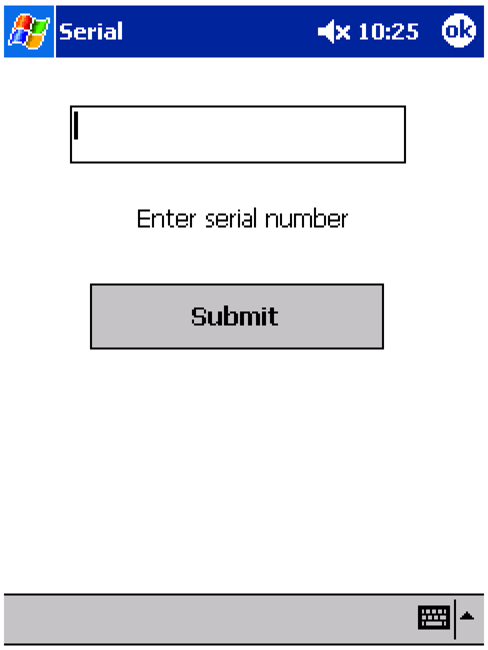


Figure 9: Serial.exe default screen

Soon after you click the Submit button, your PC should shift and focus on the section of code that we looked at earlier in IDA. You should note a little yellow arrow on the left side of the window pointing to the address of the breakpoint. At this time, right click on the memory address column and note the menu that appears. You will learn to use this menu quite frequently when debugging a program.

NOTE: The MVC is very slow when it is in execution mode if using a USB/serial connection. If you are in the habit of jumping between programs, you will quickly become frustrated at the time required for the MVC to redraw the screen. To avoid this, ensure the MVC is in break mode before changing window focus.

Before continuing, you should familiarize yourself with the various tools provided by the MVC. In particular, there are several windows you will want open while debugging. These are accessed by right clicking on the tool bar area at the top of the MVC. The three of interest are as follows:

Registers: This window lets you see the current values held by the registers. This is very useful because you can determine the registers update as the program executes.

Memory: The memory window lets you look directly in the RAM being used by the program. This is useful because the registers will often point to a memory location at which a value is being held.

Call Stack: This window lets you see the task list of the program and allows you to decipher some of the abstract commands and branches that occur in a program.

5.8 Step-through Investigation

At this point, serial.exe is loaded on the pocket PC and the MVC is paused at a break point. The next command the processor is to execute is “MOV R1, #1”. From our previous discussion on the ARM opcodes, we know that this is a simple command to move the value 1 into register 1 (R1).

Before executing this line, look at the register window and note the value of R1. You should also note that all the register values are red. This is because they have all changed from the last time the program was paused. Next hit the F11 key to execute the next line of code. After a short pause, the MVC will return to pause mode upon which time you should note several things. The first is that most of the register values turned to black, which means they did not change values. The second is that R1 now equals 1.

The next line loads the R0 register with the value in R4. Once again, hit the F11 key to let the program execute this line of code. After the brief pause, you will see that R0 is equal to R4. Step through a few more lines of code until your yellow arrow is at address 0x00011290. At this point lets take a look at the Register window.

The last line of code executed was an LDR command that loaded a value (or address representing the value) from memory into a register. In this case, the value was loaded into R1, which should be equal to 0006501C. Locate the Memory window and enter the address stored by R1 into the Address: box. Once you hit enter, you should be staring at the serial number you entered.

After executing the next line, we can see that R0 is given a small integer value. Take a second and see if you can determine its significance...OK, enough time. In R0, you should have a value equal to the number of character in the serial you entered. In other words, if you entered “777”, the value of R0 should be three, which represents the number of characters you entered.

The next line, “CMP R0, 8”, is a simple comparison opcode. When this opcode is executed, it will compare the value in R0 with the integer 8. Depending on the results of the comparison, the status flags will be updated. These flags are conveniently located at the bottom of the Registers window. Note their values and hit the F11 key. If the values change to N1 Z0 C0 O0, your serial number is not eight characters long.

At this point, serial.exe is headed for a failure message (unless you happened to enter eight characters). The next two lines of code use the results of the CMP check to determine if the value is greater than or equal to eight. If either is true, the program will jump to address 0x000112E4 where a message will be displayed on the screen. If you follow the code, you will see that address 0x000112E4 contains the opcode “LDR R1, [SP]”. If you follow this through and check the Memory address after this line executes, you will see that it points to the start of the following error message at address 0x00065014, “Incorrect serial number. Please verify it was typed correctly.”

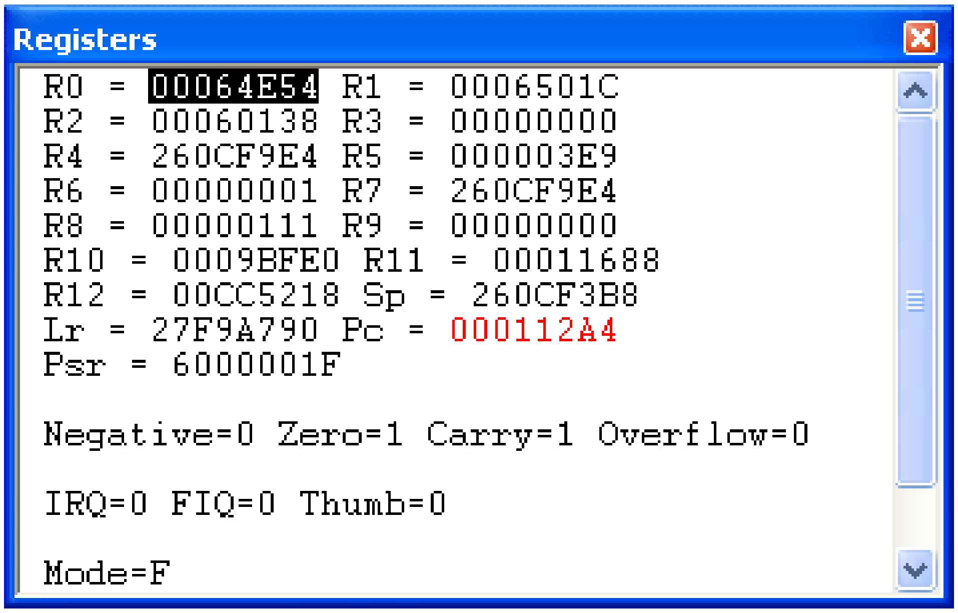
5.9 Abusing the System

Now that we know the details of the first check, we will want to break the execution and restart the entire program. To do this, perform the same steps that you previously worked through, but set a breakpoint at address 0x00011294 (CMP R0, #8). Once the program is paused at the CMP opcode, locate the Register window and note the value of R0. Now, place your cursor on the value and overwrite it with '00000008'. This very handy function of the MVC will allow you to trick the program into thinking your serial is eight characters long, thus allowing you to bypass the check. While this works temporarily, we will need to make a permanent change to the program to ensure any value is acceptable at a later point.

After the change is made, use the F11 key to watch serial.exe execute through the next couple lines of code. Continue until the pointer is at address 0x000112A4 (BL 00011754). While this command may not mean much to you in the MVC, if we jump back over to IDA Pro we can see that this is a function call to wcscmp, which is where our serial is compared to the correct serial. Knowing this, we should be able to take a look at the Registers window and determine the correct serial.

NOTE: Function calls that require data to perform their operation use the values held by the registers. In other words, wcscmp will compare the value R0 with the value of R1, which means we can easily determine what these values are. It will then return a true or false in R1.

If we look at R0 and R1, we can see that they hold the values 00064E54 and 0006501C, respectively, as illustrated by figure 10 (these values may be different for your system). While these values are not the actual serial numbers, they do represent the location in memory where the two serials are located. To verify this, place R1's value in the Memory windows address field and hit enter. After a short pause, the Memory window should change and you should see the serial number you entered. Next, do the same with the value held in R0. This will cause your Memory window to change to a screen similar to figure 11 where you should see the value '1.2.3.4.5.6.7.8', or in other words, the correct serial.



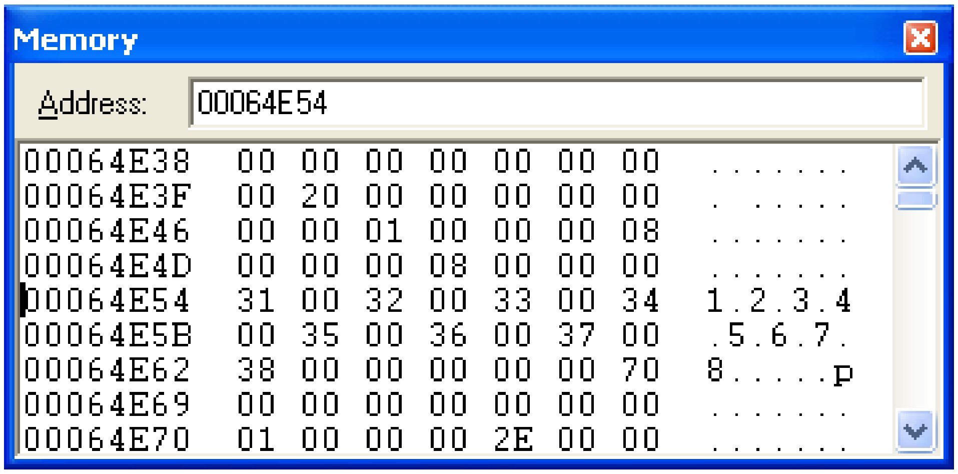
```
R0 = 00064E54 R1 = 0006501C
R2 = 00060138 R3 = 00000000
R4 = 260CF9E4 R5 = 000003E9
R6 = 00000001 R7 = 260CF9E4
R8 = 00000111 R9 = 00000000
R10 = 0009BFE0 R11 = 00011688
R12 = 00CC5218 Sp = 260CF3B8
Lr = 27F9A790 Pc = 000112A4
Psr = 6000001F

Negative=0 Zero=1 Carry=1 Overflow=0

IRQ=0 FIQ=0 Thumb=0

Mode=F
```

Figure 10: Registers window



Address: 00064E54

```
00064E38  00 00 00 00 00 00 00  .......
00064E3F  00 20 00 00 00 00 00  . .....
00064E46  00 00 01 00 00 00 08  .......
00064E4D  00 00 00 08 00 00 00  .......
00064E54  31 00 32 00 33 00 34  1.2.3.4
00064E5B  00 35 00 36 00 37 00  .5.6.7.
00064E62  38 00 00 00 00 00 70  8.....p
00064E69  00 00 00 00 00 00 00  .......
00064E70  01 00 00 00 2E 00 00  .......
```

Figure 11: Using Memory window

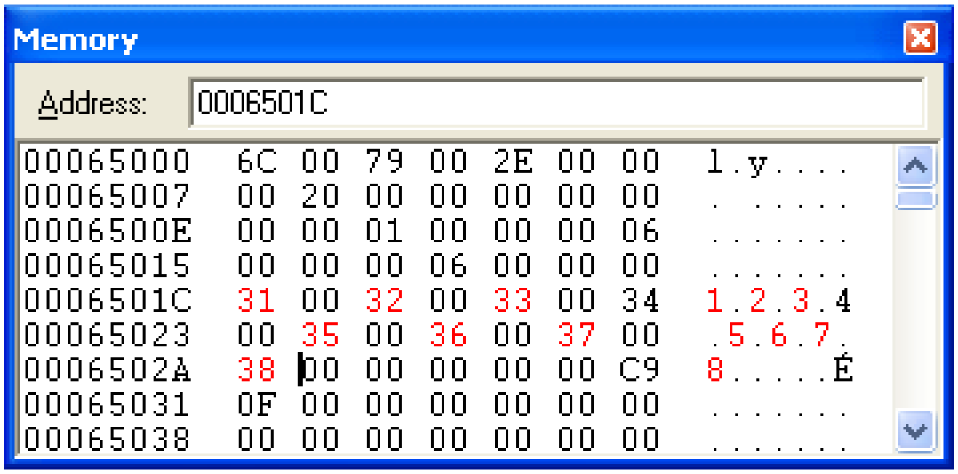
At this point a cracker could stop and simply enter the newfound value to gain full access to the target program, and then spread that serial number around. However, most serial validations include some form of dynamically generated serial number (based off of time, name, or matching registration key), which means any value determined by viewing it in memory will only work for that local machine. As a result, crackers will often note the serial number, and continue on to determine where the program can be 'patched' to bypass the protection regardless of any dynamic serial number.

Moving on through the program, we know the wcscmp function will compare the values held in memory, which will result in an update to the condition flags and R0 – R4 as follows.

R0: If serials are equal, R0 = 0, else R0 = 1.

R1: If equal, address following entered serial number, if not equal, address of failed character.
R2: If equal then R2=0, else hex value of failed character.
R3: If equal then R3=0, else hex value of correct character.

Therefore, we need to once again trick the program into believing it has the right serial number. This can be done one of two ways. The first method you can use is to actually update your serial number in memory. To do this, note the hex values of the correct serial (i.e. 31 00 32 00 33 00 34 00 35 00 36 00 37 00 38), and overwrite the entered serial number in the Memory windows. When you are done, your Memory window should look like figure 12.



```
00065000  6C 00 79 00 2E 00 00  l.y....
00065007  00 20 00 00 00 00 00  . .....
0006500E  00 00 01 00 00 00 06  .......
00065015  00 00 00 06 00 00 00  .......
0006501C  31 00 32 00 33 00 34  1.2.3.4
00065023  00 35 00 36 00 37 00  .5.6.7.
0006502A  38 00 00 00 00 00 C9  8.....É
00065031  0F 00 00 00 00 00 00  .......
00065038  00 00 00 00 00 00 00  .......
```

Figure 12: Using memory window to update values

Note: Ensure you include the 00 spacers. They are necessary.

The second method a cracker can use is to update the condition flags after the wcscmp function has updated the status flags. To do this, hit F11 until the pointer is at 0x000112A8. You should note that the Z condition flags change from Z1 (equal) to Z0 (not equal). However, if you don't like this condition, you can change the flags back to their original value by overwriting them. Once you do this, the program will once again think the correct serial number was entered. While this will temporarily fix the serial check, a lasting solution will require an update to the programs code.

Fortunately, we do not have to look far to find a weak point. The following explains the rest of the code that is processed until a message is provide on the pocket pc alerting the user to a correct, or incorrect serial number.

```
260112A8   mov        r2, #0
```

This opcode clears out the R2 register so there are no remaining values that could confuse future operations.

```
260112AC   movs       r3, r0        Moves R3 into R0 and updates the status flags.
```

In this opcode, two events occur. The first is that R0 is moved into R3. The second event updates the status flags using the new value in R3. As we previously mentioned, R0 and is updated from the wcscmp function. If the entered serial number matched the correct serial number, R0 will be updated with a 0. If they didn't match, R0 will be set to 1. R3 is then updated with this value, which is check to see if it is negative or zero.

```
260112B0   mov          r0, #1        Move #1 into R0
```

Next, the value #1 is moved into R0. While this may seem a bit odd, by moving the #1 into R0, the program is setting the stage for the next couple lines of code.

```
260112B4   movne        r0, #0        If flags are not equal, move #0 into R0.
```

Again we see another altered mov command. In this case, the value #0 will only be moved into R0 if the condition flags are *not equal* (ne), which is based on the status update performed by the previous movs opcode. In other words, if the serials matched, R0 would have been set at 0 and the Zero flag would have been set Z=1), which means the movne opcode would not be executed.

```
260112B8   ands         r3, r0, 0xFF
```

Like the movs opcode, the ANDS command will first execute and then update the status flags depending on the result. Looking at the last couple lines, we can see that R0 should be 1 if the serials DID NOT matched. This is because R0 was set to equal #1 a few lines up and was not changed by the MOVNE opcode. Therefore, the 'AND' opcode would result in R3 being set to the value of #1 and the condition flags would be updated to reflect the EQUAL status. On the other hand, if the serials did match, R0 would be equal to 1, which would have caused the ZERO flag to be set to 0, or NOT EQUAL.

```
260112BC   ldrne   r1, [sp, #8]
```

Here we see another implementation of the 'not equal' conditional opcode. In this case, if the ANDS opcode set the Z flag to 0, which would occur only if the string check passed, the ldrne opcode would load R1 with the data in SP+8. If you recall from our dissection of code in IDA Pro, you will should recall that address 0x0001125C loaded the 'correct message' into this location of memory. However, if the condition flags are not set at 'not equal' or ‘not zero’, this opcode will be skipped.

```
260112C0   mov     r0, r4             Move R4 into R0
```

An example of a standard straightforward move of R4 into R0.

```
260112C4   mov     r3, #0             Move #0 into R3
```

Another example of a simple move of #0 to R3.

```
260112C8   bne      260112F4  ;          If flag not equal jump to 0x260112F4
```

Again, we see a conditional opcode. In this case, the program will branch to 0x000112F4 if the 'not equal' flag is set. Since the conditional flags have not been updated since the 'ANDS' opcode in address 0x000112B8, a correct serial number would result in the execution of this opcode.

```
260112CC   ldr      r1, [sp, #4]         Load SP+4 into R1 (incorrect message)
```

If the wrong eight character serial number were entered, this line would load the 'incorrect message' from memory into R1.

```
260112D0   b        260112F4  ;          Jump to 0x260112F4
```

This line tells the program to branch to address 0x260112F4.
...

```
260112F4   bl       26011718  ;          MessageboxW call to display message in R1
```

The final line we will look at is the call to the Message Box function. This command simply takes the value in R1, which will either be the correct message or the incorrect message, and displays it in a Message Box.

5.10 The Cracks

Now that we have dissected the code, we need to determine how it can be altered to ensure that it will accept any serial number as the correct value. As we have illustrated, 'cracking' the serial is a fairly easy task when executing the program in the MVC by changing the register values, memory, or condition flags during program execution. However, this type of manhandling is not going to help the average user who has no interest in reverse-engineering. As a result, a cracker will have to make permanent changes to the code to ensure the serial validation will ALWAYS validate the entered serial.

To do this, the cracker has to find a weak point in the code that can be changed to bypass security checks. Fortunately, for the cracker, there is typically more than one method by which a program can be cracked. To illustrate, we will demonstrate three distinct ways that serial.exe can be cracked using various cracking techniques.

5.10.1 Crack 1: Slight of Hand

The first method we will discuss requires three separate changes to the code. The first change is at address 00011294 where R0 is compared to the #8. If you recall, this is used to ensure that the user provided serial number is exactly eight characters long. The comparison then updates the condition flags, which are used in the next couple lines to determine the flow of the program.

To ensure that the flags are set at 'equal', we will need to alter the compared values. The easiest way to do this is to have the program compare two equal values (i.e. CMP R0, R0). This will ensure the comparison returns as 'equal', thus tricking the program into passing over the BLT and BGT opcodes in the next two lines.

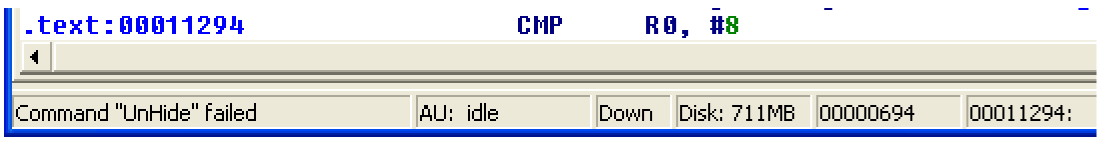
The next change is at address 0x000112B4, where we find a movne r0, #0 command. As we previously discussed, this command checks the flag conditions and if they are set at 'not equal', the opcode moves the value #0 into R0. The R0 value checked when it is moved into R3, which updates the status flags once again.

Since the movs command at address 00112AC will set Z=0 (unless the correct serial is entered), the movne opcode will execute, thus triggering a chain of events that will result in a failed validation. To correct this, we will need to ensure the program thinks R0 is always equal to '1' at line 000112B8 (ands r3, r0, #0xFF). Since R0 would have been changed to #1 in address 000112B0 (mov r0, #1), the ands opcode would result in a 'not equal' for a correct serial.

In other words, we need to change movne r0, #0 to movne r0, #1 to ensure that R0 AND FF outputs '1', which is then used to update the status flags. By doing this, the program will be tricked into validating the incorrect serial.

Changes:
```
.text:00011294              CMP     R0, #8 -> CMP R0, R0
.text:000112B4               MOVNE   R0, #0 -> MOVNE R0,#1
```

Determining the changes is the first step to cracking a program. The second step is to actually alter the file. To do this, a cracker uses a hex editor to make changes to the actual .exe file. However, in order to do this, the cracker must know where in the program file they need to make changes. Fortunately, if they are using IDA Pro, a cracker only has to click on the line they want to edit and look at the status bar at the bottom of IDA's window. As figure 13 illustrate, IDA clearly informs its user what memory address the currently selected line is located at in a the program, which can be then used in hex editor.


```
.text:00011294                 CMP     R0, #8
```
```
Command "UnHide" failed | AU:  idle | Down | Disk: 711MB | 00000694 | 00011294:
```
Figure 13: Viewing location of 0x00011294 for use in hex editor.

Once we know the addresses where we want to make our changes, we will need to determine the values that we will want to update the original hex code with. Fortunately, there are several reference guides online that can help with this. In our case, we will want to make the following changes to the serial.exe file.

IDA Addr	Hex Addr	Orig Opcode	Org Hex	New Opcode	New Hex
0x11294	0x694	Cmp r0, #8	08 00 50 E3	Cmp r0, r0	00 00 50 E1
0x112B4	0x6B4	Monve r0, #0	00 00 A0 13	Movne r0, #1	01 00 A0 13

To make the changes, perform the following procedures (using UltraEdit).

1.Open UltraEdit and then open your local serial.exe file in UltraEdit.
2.Using the left most column, locate the desired hex address.

Note: You will not always be able to find the exact address in the hex editor. You will need to count the character pairs from left to right to find the exact location once you located the correct line.

3.Move to the hex code that needs changed, and over write it.
4.Save the file as a new file, in case you made a mistake.

5.10.2 Crack 2: The Slide

The next illustration uses some of the same tactics as Crack1, but also introduces a new method of bypassing the eight-character validation, known as NOP.

The term NOP is a reference to a Non-OPeration, which means the code is basically null. Many crackers and hackers are familiar with the term NOP due to its prevalence in buffer overflow attacks. While this is outside the scope of this paper, a NOP sled (as it is often called) is used to make a part of program do absolutely nothing. The same NOP sled can be used when bypassing a security check in a program.

In our program, we have a cmp opcode that compares the length of the entered serial with the number eight. This results in a status change of the condition flags, which are used by the next two lines to determine if they are executed. While our previous crack bypassed this by ensuring the flags were set at 'equal', we can attack the BLT and BGT opcodes by overwriting them with a NOP opcode. Once we do this, the BLT and BGT opcodes will no longer exist.

NOTE: Typical NOP code is done using a series of 0x90's. This will NOT work on an ARM processor and will result in the following opcode: UMULLLSS R9, R0, R0, R0. This opcode performs an unsigned multiply long if the LS condition is met, and then updates the status flags accordingly. It is not a NOP.

To perform a NOP on an ARM processor, you simply replace the target code with a MOV R1, R1 operation. This will move the value R1 into R1 and will not update the status flags. In other words, you are killing processor time.

The following code illustrates the NOPing of these opcodes.

```
.text:00011298             BLT    loc_112E4 -> MOV R1, R1
.text:0001129C              BGT    loc_112E4 ->  MOV R1, R1
```

The second part of this crack was already explained in Crack1, and only requires the alteration of the MOVNE opcode as the following portraits.

.text:000112B4 MOVNE R0, #0 -> MOVNE R0,#1

The following describes the changes you will have to make in your hex editor.

IDA Addr	Hex Addr	Orig Opcode	Org Hex	New Opcode	New Hex
0x11298	0x698	BLT loc_112E4	11 00 00 BA	MOV R1, R1	01 10 A0 E3
0x1129C	0x69C	BLT loc_112E4	10 00 00 CA	MOV R1, R1	01 10 A0 E3
0x112B4	0x6B4	Monve r0, #0	00 00 A0 13	MOVNE r0, #1	01 00 A0 13

5.10.3 Crack3: Preventative Maintenance
At this point you are probably wondering what the point is for another example when you have two that work just fine. However, we have saved the best example for last because crack3 does not attack or overwrite any checks or validation opcodes like our previous two examples. Instead, we demonstrate how to alter the registers to our benefit *before* any values are compared.

If you examine the opcode at 0x00001128C using the MVC, you will see that it sets R1 to the address of the serial that you entered. The length of the serial is then loaded into R0 in the next line using R1 as the input variable. If the value pointed to by the address in R1 is eight characters long, it is then bumped up against the correct serial number in the wcscmp function. Knowing all this, we can see that the value loaded into R1 is a key piece of data. So, what if we could change the value in R1 to something more agreeable to the program, such as the correct serial?

While this is possible by using the SP to guide us, the groundwork has already been done in 0x0000112A0 where the correct value is loaded into R0. Logic assumes that if it can be loaded into R0 using the provided ldr command, then we can use the same command to load the correct serial into R1. This would in effect trick our validation algorithm to compare the correct serial with itself, which would always result in a successful match!

The details of the required changes are as follows.

IDA Addr	Hex Addr	Orig Opcode	Org Hex	New Opcode	New Hex
0x11298	0x68C	LDR R1, [R4, #0x7C]	7C 10 94 E5	LDR R1, [SP,#0xC]	0C 10 9D E5

Note that this crack only requires the changing of two hex characters (i.e. 7->0 & 4->D). By far this example is the most elegant and fool proof, which is why we saved it for last. While the other two examples are just as effective, they are a reactive type of crack that attempts to fix a problem. This crack, on the hand, is a preventative crack that corrects the problem before it becomes one.

6 Summary

This short example of how crackers bypass protection schemes should illustrate quite clearly the problems that programmers have to consider when developing applications. While many programmers attempt to include complex serial validation schemes, many of these eventually end up as a simple wcscmp call that can easily be 'corrected' by a cracker. In fact, the wcscmp weakness is so common that it has been called 'the weakest link' by one ARM hacker, in a nice paper available at www.Ka0s.net, which contains more than enough information to bring a complete newbie up to speed on pocket pc application reverse-engineering.

In closing, the subject of ARM reverse-engineering is somewhat new. While much has been done in the way of Linux ARM debugging, the Pocket PC OS is relatively new when compared to Intel based debugging. Ironically, the ARM processor is considered easier to debug. So, get your tools together and dig in!

References

•www.ka0s.net
•www.dataworm.net
•http://www.eecs.umich.edu/speech/docs/arm/ARM7TDMIvE.pdf
•http://www.ra.informatik.uni-stuttgart.de/~ghermanv/Lehre/SOC02/ARM_Presentation.pdf
•class.et.byu.edu/eet441/notes/arminst.ppt
•http://www.ngine.de/gbadoc/armref.pdf
•http://wheelie.tees.ac.uk/users/a.clements/ARMinfo/ARMnote.htm
•http://www3.mb.sympatico.ca/~reimann/andrew/asm/armref.pdf
•www.arm.com
•www.airscanner.com