Hackers 101

by Jason Southwell

As Delphi developers, we are afforded a lot of luxuries that many software developers don't have. We don't have to deal too much with cryptic APIs, most visual work can be done via drag and drop, and any enhancements to basic system components can easily be made by inheriting from a very strong component library that is already in the product. While these features will improve our productivity, sometimes they make us sloppy programmers. By sloppy, I mean that we get used to Delphi doing everything for us: we forget to pay attention to certain details that can eventually be devastating to our projects and, at times, reputations.

Newbies to Delphi almost always find sloppy programming causes problems at product release time, with excessive resource usage or imprudent memory management. However, even experienced developers can overlook some key aspects to their applications. Some of these oversights can certainly affect application stability, but also application security: this is the topic I'd like to discuss in this article.

We may be able to get away with these mistakes. Perhaps your application is only used in-house with a small number of trusted employees. Or perhaps it stores no sensitive data and is never operated on a system with administrative access. However, I think that a great many of us should have more concern about this issue than we generally do.

More and more we are hearing news reports of applications and websites that have been found to have security vulnerabilities. We are also hearing more and more about hackers taking advantage of these vulnerabilities. In fact, few hackers nowadays are advanced enough to be able to find new vulnerabilities themselves: most exploit well known vulnerabilities in existing applications. There are several websites that together list thousands of software vulnerabilities that a semi-skilled hacker could exploit. For a rude awakening sometime, do some surfing through www.cert.org or icat.nist.gov. These sites are great for system administrators to find and plug security holes in their installed applications, but are used everyday by hackers looking for a way into applications.

The Mind Of A Hacker

The term hacking is generally improperly defined to include hacking, cracking, and attacks. In this article, we will discuss only true hacking and how it can affect our applications. Cracking and attacks will have to wait for a later article.

Every developer and system administrator should see the movie *Sneakers*. Besides being one of my all-time favorite movies, it is generally regarded in the hacker community to be one of the best hacker movies ever. In the movie, Robert Redford leads a group of ex-hackers who are hired to break into businesses to verify security procedures. Watching this movie will clearly define what it is to be a hacker. Hacking is not crashing a computer or network. It is not about destruction, but rather the quest for free information. That's it. So if that's all that hacking is about, the real question is: could your applications be tools to be used in their quest?

For those of you who develop applications that store little or no information of interest to a hacker, don't think that you are a disqualified target. In fact, generally the hacked application is only used as a pathway to a larger source of information. In many cases, they will attempt to use your application to gain access to critical system information. This can in turn be used to gain more important and more valuable information.

Common Targets

Any type of application can contain vulnerabilities, but certainly there are characteristics that make an application a more attractive target. Some of these factors are as follows:

User Demographic: It may seem obvious that applications with the highest user exposure have the greatest potential to be encountered by a hacker. While this is true, there are other factors that are just as important to keep in mind. For example, who uses your product? A basic home user may leave open more password vulnerability than someone at a corporation whose IT department has set standards for such things. Also, it is likely that a hacker will target a user's information instead of an application's. The application vulnerabilities are only exploited to get information from or about that user. Having unconcerned users with access to sensitive information could increase the target risk of your application.

Type of Application: While any application can be a target of hacking, only those that provide a gateway into more pertinent information is generally worth hacking. If your application is an NT Service, it runs in a system process. Hacking that application would allow the hacker to have access to the NT computer just as the system process does. Hence, services are more likely to be targeted than other standard applications. Web applications are extensions to a server that run within the IIS service, which too runs as a system process and, as such, they would also be likely targets. Other than that, other applications that impersonate system accounts or store any system account information could also be targeted.

Location of Application and Code: Open source projects have really taken the computing world by storm. While they do have terrific benefits, they also create a potential security concern with

regard to hackers. Hackers who have your source code can scan through the lines of code looking for ways to exploit a compiled version of your product. This is a double-edged sword, however, because although vulnerabilities are more easily found, they can also be more quickly patched and therefore ultimately more secure. The issue here is that system administrators who use open source software must keep an active eye on these patches to close the holes as they are found. Protecting your source code from hackers' eyes will keep some vulnerabilities hidden. As they are discovered, however, they are often exploited for a much longer time before you know that the problem exists. Generally, open source projects, or at the very least projects with viewable source code, tend to be more readily targeted. In addition, if you distribute your application it can be easier to find vulnerabilities than if your application exists solely on a server under your control. Many vulnerabilities are found via trial and error. When an application is run locally, it is easier to restart it when it crashes and hence allows the hacker to work much faster. These applications will generally be easier to target than their server-only counterparts.

Common Vulnerabilities

So far, you probably have some idea of the level of threat you should expect with your application. Regardless of whether the potential threat is high or low, you should still concern yourself with plugging potential holes. Some of these holes exist in your code, but many more exist in the environment in which your application is running, or in the setup of that environment. Holes such as these have the most potential effect on your web applications and are generally the easiest for a novice or intermediate hacker to exploit.

Sniffing

When looking toward the environment for vulnerabilities, the first place to start is generally what's called 'sniffing' the application's communications. One form of sniffing, coincidentally called 'packet sniffing', examines the TCP/IP packets transmitted between the client and server portions of your application. This method is useful for viewing any data submitted from web pages or the data returned to the user on the page. Also, it could be used to monitor a direct database connection (or third tier connection) between, say, a client and its server.

Listing 1 shows the packet sniffed while connecting to an Interbase database with `IBConsole` (Natas, the free sniffing tool used to view this information, can be found at http://intex.ath.cx). As you can see from this text, we clearly see the database local path and username for making our own connection to the database. The only piece missing is the password. There are many different methods used for figuring out a password. There are tools that can be found online to perform brute force tests or encryption de-hashing of passwords found in the sniffed packet. However, it is sometimes easier to utilize social engineering to get the information. Social engineering is an amazingly easy process of tricking people into getting the information you need. It is exemplified in the whole computer-dating scene in the aforementioned movie *Sneakers*. For more information on social engineering, there is a good article at

```
www.zdnet.com/zdhelp/stories/
  main/0,5594,2669953-7,00.html
```

Sniffing can be prevented fairly easily. The solution, however, varies depending upon the communication protocol you are using.

If you wish to prevent people from sniffing your HTTP web

```
19:16:09 - 07.06.2001 Protocol: TCP  Service: unknown
Source Address: 24.30.157.68    Destination Address: 204.210.46.227
Source Port: 3371    Destination Port: 3050
45 00 00 84 0d ca 40 00 80 06 3b 92 18 1e 9d 44 cc d2     E     @   ;    D
2e e3 0d 2b 0b ea aa ae 9c c9 bb 07 80 86 50 18 44 60     .  +          P D`
69 c3 00 00 00 00 00 13 00 00 00 00 00 00 00 2c 63 3a     i             ,c:
5c 70 72 6f 67 72 61 6d 20 66 69 6c 65 73 5c 66 69 72     \program files\fir
65 62 69 72 64 5c 62 69 6e 5c 61 72 63 61 6e 61 73 69     ebird\bin\arcanasi
74 65 2e 67 64 62 00 00 00 20 01 1c 06 53 59 53 44 42     te.gdb       SYSDB
41 3c 00 1e 0b 51 50 33 4c 4d 5a 2f 4d 4a 68 2e 3a 04     A<   QP3LMZ/MJh.:
3c 00 00 00 3e 00                                         <   >
-------------------------------------------------------------------------------
```

➤ *Listing 1*

applications, the easiest thing to do is run it on an SSL secured server. This is the same technology that all secure websites use. Due to the processing requirements of an SSL encrypted connection, you generally would only use it for pages that pass sensitive data to and from the user, lest you sacrifice some degree of scalability of the application.

If you use straight TCP/IP sockets to communicate instead of HTTP, you have a couple of options. If your application is run in a fairly closed environment with only a few installations, and cost is not a factor, a very secure alternative is set up a virtual private network (VPN) between all of your users. This, of course, is not practical in an application geared toward the general populace, and greatly decreases some of the basic advantages that the net brings to you. In that case, you can use one of the many encryption algorithms available to encode your communication. There are many Delphi components out there to help with that. As encryption is not really the focus of this article, a simple search at www.vclcrawler.com with the word encryption will yield almost 100 results and give you a good starting point.

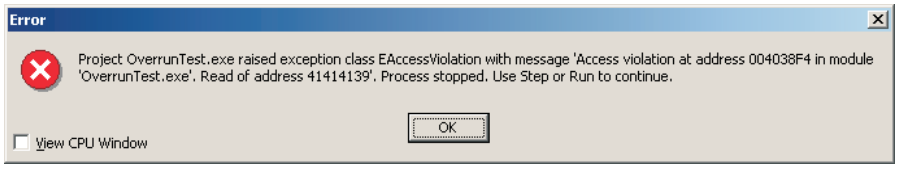
Impersonation

Web applications are more vulnerable in this area due to their stateless nature. Unless you want your users to re-enter logon information for every page they browse, state must be passed in every call to the server. These applications have no way of knowing that a user has finished with the website. Usually a web developer will program a timeout into the application to automatically log the user

off when the page hasn't been hit after some suitable time interval. While this takes care of the problem to some extent, there is that brief period of time where a hacker could take control of the user's session to access the site. Also, hackers that have direct access (or obtain such access) to a user's computer can sit down and press the Back button to see the pages the user had just seen.

So what can be done in addition to the timeout to help solve this problem? One easy fix that can have some benefit is to add an expiration tag in the header of each of your web pages (see Listing 2). This will prevent someone from sitting at the computer of a user who had just accessed a sensitive web application and using the Back button to view cached pages. When Back is pressed, the browser will attempt to refresh the data and the page will be resubmitted. Sessions that have been timed out will require a re-login at this point. This unfortunately does not solve the problem completely, but rather makes it just a bit more difficult for a would-be hacker. Other solutions are much more complex and, in some cases, not adequate either. For example, you could hide a Java applet in your page that repeatedly contacts a server to report the open browser. This would guarantee that if the browser is closed, or if the user switches websites, the session would be closed. A method like this does not solve the problem of a browser window left open by a user who went to lunch, but your timeout would eventually take care of that. Also, it would require additional server power to manage the Java pings. To the best of my knowledge, there is not a perfect solution to this when using HTML as the client interface. If you have suggestions, please feel free to send me an email.

➤ *Figure 1*



➤ *Above: Listing 2*

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<META HTTP-EQUIV="expires" CONTENT="0">
<html>
<head>
  <title>This is the Page Title</title>
</head>
<body>
This Page Expires Now!
</body>
</html>
```

➤ *Below: Listing 3*

```
var
  Buf1 : array[0..15] of Char;
  pc : PChar;
  s : string;
begin
  pc := @Buf1;
  InputQuery('Buffer Overrun Test','Enter some data here',s);
  StrPCopy(pc,s);
  Showmessage(Buf1);
end;
```

Overruns

Now down to the vulnerability that has the potential to cause the most damage. Buffer overruns have been getting a lot of attention lately as more and more applications are found to have major vulnerabilities. A buffer overrun found in your software could allow a hacker to execute code giving him administrative control of your software or, worse, control of the computer your software is running on. The worst part is that this vulnerability is not limited only to web applications. NT services, web applications, or any generic application that runs in a system or administrator context, could have huge holes ready to be exploited.

Let's look into what happens with a buffer overrun. When a program allocates a chunk of memory of a specific size and subsequently stuffs too much data into it, a buffer overrun has occurred. The extra data placed into the buffer inadvertently (or intentionally as we'll see in a bit) overwrites some information critical to normal program execution. This will most likely manifest itself as an access violation or page fault in the program. Maybe you've seen one of those before.

Listing 3 shows some example code expecting user input. When you execute the code a dialog box pops up prompting the user for input. If the user enters a response less than 15 characters, all is well. However, if the user answers with a string greater than 15 characters, the application has not prepared for it and a buffer overrun will occur. Depending upon what data has been overwritten in memory, it may or may not cause an access violation. Just for a test, enter a string with 100 A characters in it. You will see an access violation that looks similar to Figure 1.

In machine code there is a special register called EIP. This stands for the Extended Instruction Pointer. This pointer is what directs the processor to the correct line in which to process next. Placing enough As in that string will eventually overwrite that EIP. In the message shown in Figure 1, you will see the address $41414139. Interestingly enough, #41 is the ordinal value of A. What we have done with our buffer overrun is told the EIP that the next instruction to execute will be found at the memory address $41414139, or rather the memory address AAAA9.

This is key to being able to exploit a buffer overrun. If the EIP can be overwritten, then the intelligent hacker can overwrite it with an address that he is familiar with. In fact, if he could only place code at that memory address to be executed, then he would be in control.

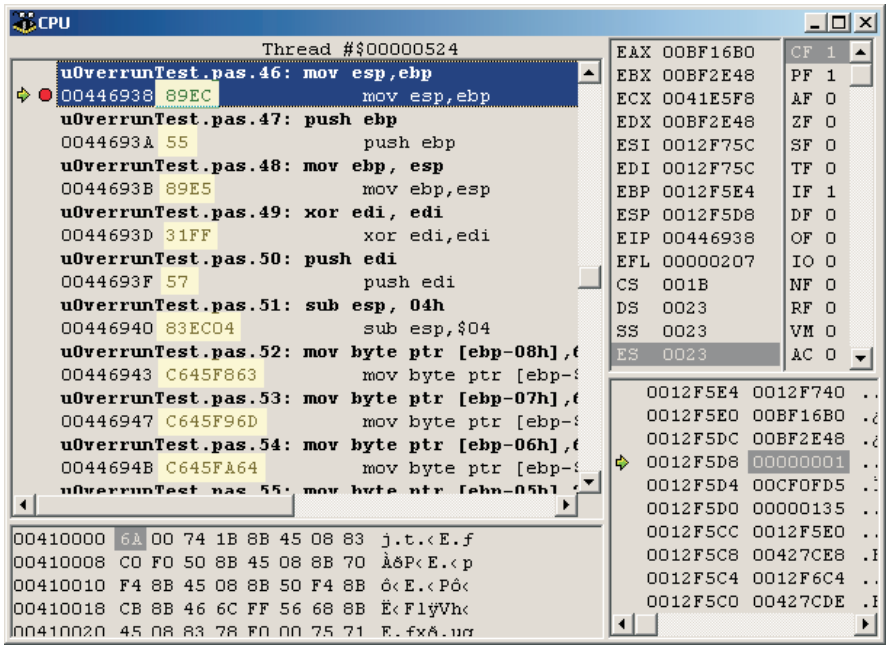
Well, in fact, he can. Listing 4 shows some assembly code. This code when executed will simply open up the command prompt.

That doesn't sound too interesting until we learn that the command prompt is opened up in the security context of the application that calls the code. Therefore, if there were only some way to place this code into an already running service or web application, we could have system-level access to the computer.

Well, guess what: we can. Listing 4 shows the assembler code in the process of being debugged using Delphi's CPU window. The highlighted numbers show what are known as machine op-codes. These op-codes are what are actually being processed by the CPU and stored at the address pointed by the `EIP`. In our case, we need to place these op-codes into the string we pass in our buffer overrun exploit. All that is needed now is to pad the string with enough characters to push the pointer of string's starting position out to the appropriate memory address for overwriting the `EIP`. This will then execute our assembler code and open the command prompt in the same security context as the hacked application.

There is actually a lot more to understand about exploiting overruns before you can be a fully-fledged hacker, but that is the basics of the process they go through to do it.

And now for the good news... Delphi applications are generally not too vulnerable to buffer overrun hacks. There are two simple reasons.



➤ *Figure 2*

First, overruns happen when a variable that was expecting a value of a certain size is filled with a value of a larger size. Usually, in the process of writing a Delphi application, we use `String` variables to house this type of information. `String`s are dynamic in size and Delphi manages that size for us. This means that if the value is expected to have 16 characters but 100 are submitted, then there is really no problem because Delphi automatically makes the adjustment to our variable for us.

Secondly, there are basically two places that can be used to store information for a running application: the stack and the heap. Overruns can be performed on both, but with stack overruns it is much easier to overwrite the `EIP` and hence run code of your choice. In Delphi as in other languages, local variables are stored on the stack and all other variables on the heap. However, in Delphi, dynamic variables are always stored in the heap with local dynamic variables referenced in the stack as a pointer to the heap. This means that to violate the stack, the variable would have to be one of just a few types, primarily a static array.

The safest bet in a Delphi application is to use `String` variables instead of `PChar`s or arrays of `char`s. While it is true that `PChar`s would be placed on the heap instead of the stack, they still can be exploited. Heap overruns are very similar to stack overruns; however, finding a function pointer to overwrite is somewhat harder. To be completely safe, don't use `PChar`s or other dynamic memory variables unless absolutely necessary.

Obviously, there are times when you must use `PChar`s or dynamic arrays in your applications. In fact, many DLLs or other APIs that you might integrate with will use `PChar`s to transfer string data in the procedure calls. These variables can still safely be used if you use careful consideration when assigning their value. Be sure to check the length of all values being placed into the variable. If they are too long, truncate or abort the operation.

➤ *Listing 4*

```
asm
  mov esp,ebp
  push ebp
  mov ebp, esp
  xor edi, edi
  push edi
  sub esp, 04h
  mov byte ptr [ebp-08h],63h
  mov byte ptr [ebp-07h],6Dh
  mov byte ptr [ebp-06h],64h
  mov byte ptr [ebp-05h],2Eh
  mov byte ptr [ebp-04h],65h
  mov byte ptr [ebp-03h],78h
  mov byte ptr [ebp-02h],65h
  mov eax, $78019B4A // Set this to the correct offset for the
    // System() function in your msvcrt.dll version
  push eax
  lea eax, [ebp-08h]
  push eax
  call dword ptr [ebp-0ch]
  mov eax,[ebp-$08]
  push eax
  call FreeLibrary
end;
```

This is somewhat difficult if you are calling out to another DLL. You probably have no idea what length to limit the `PChar` you will pass to it. If the developers of the DLL did their job, they will check this themselves for you; however, you cannot be guaranteed that this is the case. A good precaution is to search one of the many vulnerability databases online, especially if the API or DLL is rather well known or widely used. These databases keep fairly up to date and, as vulnerabilities are found, they are quickly documented.

These overrun concepts aren't easy to get on the first try and a true grasp on the subject would most certainly require more study. For more information about buffer overruns, you might want to check the *Tao Of Windows Buffer Overflow* at www.cultdeadcow.com/cDc_files/cDc-351/.

Keep Informed

As I'm writing this I'm sure there is some hacker out there coming up with some cool new way to manipulate your code. As developers, we will be forever working to catch up with the hackers: after all, by definition, they are trying to learn about our programs and the information contained within them. We will try to patch their exploits and they will try to exploit those fixes. It's a vicious circle that simply ends with the conclusion that there will never be a 100% security guarantee. If you do decide to advertise such a guarantee, than be prepared that hackers everywhere will target you. They love a good challenge.

The best thing that we as developers can do is to keep ourselves informed about the latest hacking techniques. Even if we ourselves never intend to hack into some government database or break into the Federal Reserve, we should have some idea of the techniques used to do it. Only through this knowledge will we be able to be proactive at securing our own applications from this type of breach.

There are several good resources that you should be aware of to help you stay informed. First of all, everyone should sign up to receive *2600 The Hacker Quarterly* (www.2600.com). This magazine is put out by 2600 (a worldwide hacker organization) and is created together by hackers of all ages and skill levels. Reading it will really get you tuned into how hackers think: remember that understanding is key to defending.

Vulnerabilities are everywhere and with the hacking mindset being what it is, there will continue to be more discovered for a very long time. To end on a quote from that great movie *Sneakers* that sums up the hacker mindset: 'Too many secrets'.

Jason Southwell is President and CEO of Arcana Technologies Incorporated. He has over 10 years experience managing, designing and developing internet technology and computer software and has been directly responsible for several ERP and internet B2B projects. He has been a Delphi developer since Delphi 1.0 way back in 1995. You can contact him at jason@southwell.net