CHAPTER 23

Writing Exploits that Work in the Wild

Every bug has a story. A bug is born, lives, and then dies, often without ever being discovered or exploited. For a hacker, each bug is a golden opportunity to create an exploit, a magic spell that turns any vulnerable wall into a door. But it's one thing to create a spell that works in the lab and a completely different thing to create one that works in the electric jungle that is the modern Internet. This chapter focuses on creating exploits that can be used successfully in the wild.

Factors in Unreliability

This section of the chapter covers the various reasons your exploit may not work reliably in the wild. Keep in mind that although there are many reasons for your exploit not to work, as Anakata says, "even a blind chicken finds a seed occasionally."

Magic Numbers

Some vulnerabilities, such as the RealServer stack overflow described in Chapter 17, lend themselves to reliable exploitation. Others, such as the `dtlogin` heap double-free, are nearly impossible to reliably exploit. However, it's impossible to know how reliable you can make a given exploit until you try it.

In addition, exploiting more and more difficult vulnerabilities is the only way in which to learn new techniques. Merely reading about a technique will never truly give you the essential knowledge of how to use that technique. For these reasons, you should always make the extra effort to make your exploits as reliable as possible. In some instances, you'll have a perfectly good exploit that works 100 percent of the time in the lab but only 50 percent of the time in the wild, and you'll find yourself having to rewrite it from scratch in order to improve its reliability in the real world.

When you create the first exploit for a particular vulnerability, the exploit might work only on your machine. If so, you probably hardcoded some important things into it, most likely the return address or `geteip` address. When you can overwrite a function pointer or a stored return address, you will need to redirect execution somewhere—this location is likely dependent on many factors, only some of which you control. For the purposes of this chapter, we will call this situation a *one-factor exploit*.

Likewise, you may have a place in your exploit where a pointer to a string is located. The target program uses this pointer before you get control of execution. In order to effectively gain control, you may need to set that pointer (part of your attack string) to a harmless place in memory. This step would add an additional factor that you'd have to know in order to successfully exploit your target.

Most easy exploits are one- or two-factor exploits. A basic remote stack overflow, for example, is usually a one-factor exploit—you only have to guess the address of your shellcode in memory. But as you progress to heap overflows, which are typically two-factor exploits, you must begin to look for ways to help narrow the level of chaos in the system.

Versioning

One important problem you will face when running an exploit in the wild is that you will rarely know what is loaded on the other machine. Perhaps it is a Windows 2000 Advanced Server box, just like the one in your lab, or perhaps it is running ColdFusion and moving memory all around. Or perhaps it is loaded with Simplified Chinese Windows 2000. It might have patches that are above and beyond the latest service pack; some of them might have been installed manually or even mis-installed. It is also possible that the remote system is a Linux box running on an Alpha, or that the system is an SMP box, which may affect the way in which the server you are attacking runs. Many of the public Microsoft RPC Locator exploits fail on SMP boxes, or boxes loaded with Xeon processors, which Windows thinks are dual processor boxes. These sorts of issues are very difficult to track remotely.

In addition, when you run heap corruption exploits, you will have problems keeping other people from using the service while you are corrupting the heap. Another common issue with heap overflows is that they overwrite function pointers that are dependent on a particular version of libc. Because each version of Linux has a slightly different libc, this means the exploit must be specific to certain distributions of Linux. Unfortunately, no one distribution of Linux has a majority of the market share, so it is not as easy to hardcode these values into exploits as it is with Windows or commercial Unix boxes.

Keep in mind that many vendors release multiple versions of Unix under the same version number. Your Solaris 8 CD will be different from another Solaris 8 CD, depending on when each was purchased. Your version may have patches that another CD does not, and vice versa.

Shellcode Problems

Some programmers spend weeks writing their shellcodes. Others use packaged shellcodes from Packetstorm (`http://packetstormsecurity.org/`). However, no matter how sophisticated your shellcode, it is still a program written in assembly language and executing in an unstable environment. This means that the shellcode itself is often a point of failure.

In your lab, you and your target box are located on the same Ethernet hub. In the wild, however, your target could be on a different continent, under the control of someone who has set up their own network according to their own whims. This may mean that they have set their MTU to 512, that they are blocking ICMP, that they have port-forwarded IIS to their Windows box from their Linux firewall, or that they have egress filtering on their firewall or some other complication.

Let's divide shellcode problems into the following categories.

Network Related

MTU (Maximum Transmission Unit) or routing issues can be a problem when you run your shellcode. Sometimes you'll attack one IP, and it will call back to you from a different IP or interface. Egress filtering is also a common problem. Your shellcode should close cleanly if it cannot get back to you because of filtering. You may want to include UDP or ICMP callback shellcode.

Privilege Related

In Windows, a particular thread can be running without the privileges it needs to load `ws2_32.dll`. The usual solution is to steal the thread you came in on,

assume `ws2_32.dll` is already loaded, or call `RevertToSelf()`. On some versions of Linux (SELinux, and so on) you may run into the same kind of privilege issues.

In some rare cases, you will not be allowed to make socket connections or listen on a port. In these cases, you may want to modify the original program's execution flow (for example, disable the target process's normal authentication to allow yourself to manipulate it further, modify a file it reads from, add to a userlist, and so on) or find some way for your shellcode to leverage its access without outside contact.

Configuration Related

Misled OS identification may cause to you put the wrong shellcode or return addresses into the exploit. It's difficult to remotely determine Alpha Linux from SPARC Linux; it may be wise simply to try both.

If your target process is `chrooted`, `/bin/sh` might not exist. This is another good reason not to use standard `exeve(/bin/sh)` shellcode.

Sometimes the stack base will change based on which processor you are attacking. Also, not all instructions are valid on every type of processor. Perhaps your target has an old Alpha chip, for example, and you tested your shellcode only on a new one. Or, perhaps the new SGI machine you are attacking has a large instruction cache that is not cleared during your attack.

Host IDS Related

chroot, LIDS, SELinux, BSD `jail()`, gresecurity, Papillion, and other variations on a theme can cause problems for your shellcode at many levels. As these technologies become more popular, expect to deal with them from within your shellcode. The only way to know whether they will affect you is to install them and test them yourself.

Okena and Entercept both hook system calls and do profiling based on which system calls that application is normally seen to do. Two ways to defeat this profiling are to model the application's normal behavior and try to stay somewhat within that, or to try to defeat the system call hooking itself. If you have a kernel exploit, now is your chance to use it directly from your shellcode.

Thread Related

In heap overflows, another thread may wake up to handle a response, try to call `free()` or `malloc()`, and then crash the process when it finds the heap corrupted.

Another thread may be watching your thread to discover whether it completes in time. Because you have taken over your thread, it may kill you off to recover the process. Check for heartbeat threads from your shellcode and try to emulate their signals if possible.

Your exploit may be relying on return addresses that are only valid for one thread, which is usually due to poor testing on your part.

Countermeasures

There are many ways in which your exploit attempt can become unstable or fail to work altogether. However, there are also many ways in which you can compensate for these problems. It's important to remember that you are writing an application that should never have existed—an exploit. Exploits exist only because of bugs in other software. Hence, creating reliable exploits is not a matter of simply using software engineering. At all times within the process, you should be constantly trying to find alternative methods of solving problems that crop up. When in doubt, think: "What would John McDonald do?" Here is an excerpt from *Phrack* (number 60, December 2002) which outlines his philosophy. Keep his words in mind whenever you run into trouble.

> *PHRACKSTAFF: You have found quite a lot of bugs in the past and developed exploit code for them. Some vulnerabilities required new creative exploitation concepts which were not known at that time. What drives you into challenging the exploitation of complicated bugs and what methods do you use?*
>
> *John McDonald: Well, my motivations have definitely changed over time. I can come up with several ancillary reasons that have driven me at different times during my life, and they include both the selfish and the altruistic. But, I think it really comes down to a compulsion to figure all this stuff out. As far as methods, I try to be somewhat systematic in my approach. I budget a good portion of time for just reading through the program, trying to get a feel for its architecture and the mindset and techniques of its authors. This also seems to help prime my subconscious.*
>
> *I like to start at the lower layers of a program or system and look for any kind of potential unexpected behavior that could percolate upwards. I will document each function and brainstorm any potential problems I see with it. I will occasionally take a break from documentation, and do the considerably more fun work of tracing back some of my theories to see if they pan out.*
>
> *As far as writing exploits, I generally just try to reduce or eliminate the number of things that need to be guessed.*

When your exploit is almost complete, but seems not to progress, become someone else. Write your exploit "Halvar"-style—spend a great deal of time in IDA Pro examining in detail the exact location of failure and everything the program does from then on. Thrash at it madly with super-long strings. Examine what the program does when it's not dying because of your exploit. Perhaps you can find another bug that will be more reliable.

It's often useful to learn about exploitation techniques used on platforms other than those you are familiar with. Windows techniques can come in handy on Unix, and vice versa. Even when they don't come in handy, they can provide a needed inspiration for what your final exploit needs in order to be successful.

Preparation

Always be prepared. In fact, always have a stack of hard drives available with every OS in every language on them, with every service pack and patch available, and be prepared to cross reference addresses among them to determine which set of addresses works on all your targets. VMWare is a great help in this case, although VMWare and OllyDbg occasionally don't get along, which can be troublesome. Cross-referencing the database of all possible addresses can cut down your brute forcing time as well.

Brute Forcing

Sometimes the best way to make your exploit robust is to exhaust the range of possible magic numbers. If you have a huge list of potential `return to ebx` addresses, perhaps you should simply run through them all. In any event, brute forcing is often a last resort, but it is a perfectly valid last resort.

There are, however, a few tricks that can keep you from wasting time and leaving more logs than you need to. Determine whether you can check more than one address at a time while you are brute forcing. Cache any valid results so that you can check for those first. Machines on any given network tend to all be set up in the same way, so if your technique worked once, it will probably work again.

Sending ludicrously large shellcode buffers sometimes can give you a reasonable chance of hitting your magic number correctly as well. And if possible, try to keep your magic numbers related. If you know that one address you'll need will always be near another, you will be much better off than when they are completely independent of each other.

Memory leaks can often make brute forcing much easier. Sometimes you don't even need a real memory leak in order to fill up memory with your shellcode. For example, in CANVAS's IIS ColdFusion exploit, we make 1,000

connections to the remote host, each of which sends 20,000 bytes of shellcode and NOPs. This procedure quickly fills up memory with copies of the shellcode. Finally, without disconnecting any of our other sockets, we send the heap overflow. It must guess the location of our shellcode, but it nearly always guesses correctly because most of the process's memory is filled up with it.

Filling up a process's memory is easy when the process is multithreaded as is IIS. Even when the process is not multithreaded, a memory leak can accomplish nearly the same thing. And if you can't find a memory leak, you may find a static variable that holds the last result of your query and is always in the same place. If you look at the entire program to see whether it has any operations you can manipulate to accomplish this sort of goal, you will almost always find something useful.

Local Exploits

There is no reason to have an unreliable local exploit. When you map yourself into the process space, you control nearly everything—the memory space, signaling, what's on the disk, and the location of the current directory. Many people create more problems than they need to with local exploits; it's the sign of a beginner to have a local exploit that doesn't work every time.

For instance, when writing a simple Linux/Unix local buffer overflow, use `exeve()` to specify the exact environment for your target process. Now, you can calculate exactly where in memory your shellcode will be, and you can write your exploit as a return-into-libc attack without any guesswork. Personally, we like to return into `strcpy()` and copy our shellcode into the heap and then execute it there. We can use `dlopen()` and `dlsym()` to find `strcpy()`'s address during the exploit. This sort of sophistication will keep your exploits working in the wild.

As pointed out by Sinan Eren (known more widely as *noir*), when attacking the kernel, you can map memory to any location needed, making it possible to set the return address to the exact place your shellcode begins, even if you can only use one character to which to return. (In other words, `0x00000000` can be a perfectly valid return address when you're writing a local kernel attack.)

OS/Application Fingerprinting

For many purposes, the fingerprints that tools like Nmap or Xprobe can provide are only one part of the picture. When you exploit an application, you need to know more than just what operating system you are targeting. You also need to know the following:

- Architecture (x86/SPARC/other)
- Application version

- Application configuration
- OS configuration (non-exec stack/PaX/DEP and so on)

In many other cases, OS identification is completely useless, because you are being proxied from one host to the next. Or, perhaps you simply don't want to send bizarre OS identification packets to the host because that would fingerprint you to any listening network IDS. Therefore, to write reliable exploits, you must often find unique ways to fingerprint your remote host that lie within the bounds of completely normal traffic.

It's always best to be able to do your fingerprinting against the same port that you eventually will be attacking. The following example is used in CANVAS's MSRPC exploit. You can see that simply by using port 135 (the targeted service) we can finely narrow in on which OS we are targeting. First, we split XP and Windows 2003 from NT 4.0 and Windows 2003. Then we split 2003 from XP (using another function not shown here). Then we split Windows 2000 from NT 4.0. This entire function uses publicly available interfaces on port 135 (TCP), which is good because this may be the only open port. Using this technique, our exploit can narrow its targeting down to the correct platform with only a few simple connections.

```
    def runTest(self):
        UUID2K3="1d55b526-c137-46c5-ab79-638f2a68e869"
        callid=1
        error,s=msrpcbind(UUID2K3,1,0,self.host,self.port,callid)
        if error==0:
            errstr="Could not bind to the msrpc service for 2K3,XP - assuming
NT 4 or Win2K"
            self.log(errstr)
        else:
            if self.testFor2003(): #Simple test not shown here.
                self.setVersion(15)
                self.log("Test indicated connection succeeded to msrpc service.")
                self.log("Attacking using version %d:
%s"%(self.version,self.versions[self.version][0]))
                return 1

        self.setVersion(1) #default to Win2K or XP
        UUID2K="000001a0-0000-0000-c000-000000000046"
        #only provided by 2K and above
        callid=1
        error,s=msrpcbind(UUID2K,0,0,self.host,self.port,callid)
        if error==0:
            errstr="Could not bind to the msrpc service for 2K and above -
assuming NT 4"
            self.log(errstr)
            self.setVersion(14) #NT4
```

```
        else:
            self.log("Test indicated connection succeeded to msrpc service.")
            self.log("Attacking using version %d:
%s"%(self.version,self.versions[self.version][0]))
            return 1  #Windows 2000 or XP

        callid=0
        #IRemoteDispatch UUID
        UUID="4d9f4ab8-7d1c-11cf-861e-0020af6e7c57"
        error,s=msrpcbind(UUID,0,0,self.host,self.port,callid)
        #error is reversed, sorry.
        if error==0:
            errstr="Could not bind to the msrpc service necessary to run the
attack"
            self.log(errstr)
            return 0
        #we assume it's vulnerable if we can bind to it
        self.log("Test indicated connection succeeded to msrpc service.")
        self.log("Attacking using version %d:
%s"%(self.version,self.versions[self.version][0]))


        return 1
```

Information Leaks

We are past the era in which every exploit was simply a fire-and-forget missile. These days, a good exploit writer looks for ways in which to guide his attack directly to the target. There are methods for obtaining information, often specific memory addresses, from your targets. We list some of these here:

- Reading and interpreting the data the target sends to you. For example, MSRPC packets often contain pointers that are marshalled directly from memory. These pointers can be used to predict the memory space of your target process.
- Using heap overflows to write into your data before it is sent back to you can tell you where in memory your buffer is.
- Using `frontlink()`-style heap overflows to write the address of `malloc` internal variables into your data before it is sent back to you can tell you where in memory malloc's function pointers are via a simple calculation.
- Overwriting a length field can often allow large parts of server memory to be sent to you (think BIND TSIG overflow).

- Utilizing an underflow or other similar attack can allow parts of server memory to be sent to you. FX of Phenoelit uses this method successfully with echo packets for his Cisco HTTPD exploit. His work is a stellar example of combining two exploits to produce one very reliable exploit.

Looking at timing information can be a valuable way in which to gain insight into what kinds of errors your exploit is running into. Did it send you a reset packet right away, or did it time out and then send you a reset?

Halvar Flake once said, "No good hacker just looks for one bug." An information leak can make even a difficult bug possible. Even PaX (the advanced kernel-based memory protection patch) is easily defeated with a good enough information leak.

Conclusion

Say that you are writing an exploit for a custom Win32 Web server. After a day's work, the exploit, a simple stack overflow, works perfectly five times out of six. It uses a standard "overwrite the exception handler structure" technique that points into the processes memory space. This in turn points to a `pop pop return` in a `.text` segment. However, because the target process is multithreaded, occasionally another thread overwrites the shellcode, and the attack fails. So you rewrite the exploit using a much smaller string, which allows the original function to return safely, and eventually obtain control via a saved return pointer in a stack frame a few returns away. This technique, although limiting the size of the shellcode you could use, is much more reliable.

The point here is that sometimes you can't rely on even very stable techniques—sometimes you must test several different methods of exploiting a bug and then try each method on however many test platforms you can until you find the best solution. When you are stuck, try making your attack string extra long or as short as possible, or injecting characters that may cause something different to happen. If you have the source code, try painstakingly following your data as it flows through the program. Overall, don't give up. You must have a large amount of self-confidence to stay in this game, because until your exploit finally works, you will never know whether you will be successful.

We assure you, your persistence is worth it. But you must become comfortable with the fact that sometimes you will never know why your exploit does not work in the wild.