Editors: Elias Levy, aleph1@securityfocus.com
Iván Arce, ivan.arce@coresecurity.com

The Shellcode Generation

"Now, when your weapons are dulled, your ardor damped, your strength exhausted and your treasure spent, other chieftains will spring up to take advantage of your extremity. Then no man, however wise, will be able to avert the consequences that must ensue."

— *Sun Tzu,* The Art of War

Iván Arce
Core Security Technologies

Attackers carry out many network security compromises using exploitation programs, or exploits, which take advantage of bugs in software running on vulnerable systems. These programs are often the only remaining evidence of a security compromise; by analyzing them, we can assess the incident's impact and the attackers' skills and intent. We can build an entire taxonomy of attacks by understanding these programs' technical capabilities and their connection to those who develop and use them.

For the past decade, exploit tools have signaled the evolution of a community of adversaries comprising numerous inexperienced and unskilled "ankle biters"—commonly referred to as script kiddies—and a few experienced, technically savvy attackers. Exploit programs are telltale signs of attackers' sophistication: if studied meticulously, they can provide insight into current and future trends.

Dissecting exploit code

Webster's dictionary defines the verb *exploit* as "to use or manipulate to one's advantage." In the context of information security, we translate this to mean taking advantage of a vulnerable system in a way that subsequently affects the system's security. Recognizing that attackers use exploit code as a weapon, we must understand how exploits work and what they're used for.

The simplest form of exploit program is known as the *proof-of-concept* (POC) *exploit*. Its only goal is to demonstrate without a doubt that a security flaw exists, often by causing the vulnerable program to malfunction in a noticeable manner, such as terminating prematurely or abnormally. To prove not only that a given software bug exists but also that attackers could exploit it for specific purposes, the writer of a POC exploit generally turns to what software vendors and security researchers refer to as "execution of arbitrary code on the vulnerable system" to demonstrate that an outsider can execute commands on affected systems.

Exploit tools are artifacts that let attackers fulfill their intentions beyond simply demonstrating that a software flaw exists. From the exploit developer's viewpoint, an exploit must be able to use a given vulnerability to achieve a specific goal, while coping with the vulnerable system's operational characteristics, including network topology, running environment, and security countermeasures.

Studying exploits furnished by researchers or found "in the wild" on compromised systems can provide valuable information about the technical skills, degree of experience, and intent of the attackers who developed or used them. Using this information, we can implement measures to detect and prevent attacks. (Note that those who use an exploit are not necessarily the designers or developers. This becomes evident when attackers' actions during a network security compromise are not on par with the experience and technical knowledge required to build the exploits used.)

From a functional perspective, exploits have three clearly distinguishable components: the attack vector, exploitation technique, and exploitation payload.

Attack vector

An attack vector is the mechanism the exploit uses to make a vulnerability manifest. With software flaws, it's the series of actions required to reach and trigger the buggy portion of the program.

A software bug that illustrates this concept is the Secure Sockets Layer Private Communications Technology (SSL PCT) vulnerability discovered by Mike Down and Neel Mehta of Internet Security Systems' X-Force team (http://xforce.iss.net/xforce/alerts/id/168). The bug is a fairly common buffer overflow condition in the Microsoft library that implements the SSL protocol;

it's used in several Microsoft software packages, including the Web server implementation in the Internet Information Services (IIS) package.

In a detailed analysis, Core Security Technologies' Juliano Rizzo found that seven different network services can reach and trigger the vulnerable code in many Windows programs using an equal number of TCP ports (www.securityfocus.com/archive/1/361836). This is one software bug with seven known attack vectors.

Similarly, other researchers at Core Security Technologies found numerous attack vectors for the slew of vulnerabilities in the Windows OS components targeted by the Blaster and Sasser worms of 2003 and 2004 (www.corest.com/common/showdoc.php?idx=393&idxseccion=10). Like most exploits, however, each worm used only one attack vector.

In response to the hardening of operating systems (decreasing the number of services exposed to attack) and security mechanisms such as filtering firewalls and proxies (restricting connectivity), we should expect increasingly sophisticated exploit programs to use more than one—or even all—available attack vectors. Such exploits will more effectively target systems that operate under different configurations and operational environments.

Exploitation technique

An exploitation technique is the algorithm that exploits use to alter a vulnerable program's execution flow and thus yield control to the attacker. To exploit a software bug, an attacker must not only find and use a valid attack vector but also devise a suitable technique for modifying the execution flow and running the attacker's commands on the system.

In "Beyond Stack Smashing: Recent Advances in Exploiting Buffer Overruns," (*IEEE Security & Privacy,* July/Aug., pp. 20–27), Jonathan Pincus and Brandon Baker provided a comprehensive account of past and current techniques for exploiting software flaws. As the classic introduction to exploitation techniques for buffer overflow vulnerabilities, they cited AlephOne's "Smashing the Stack for Fun and Profit" (www.phrack.org/show.php?p=49&a=14) and DilDog's "The Tao of Windows Buffer Overflows" (www.cultdeadcow.com/cDc_files/cDc-351/). The techniques presented in those two articles are already being applied to exploiting other forms of software bugs.

Several information security researchers have refined, improved, and even superseded these techniques since their publication nearly a decade ago. The results are evident in myriad research reports and in exploits found in the wild on compromised systems. Advancements in exploit techniques and countermeasures are testimony to attackers' and defenders' continuing attempts to dull their adversaries' weapons.

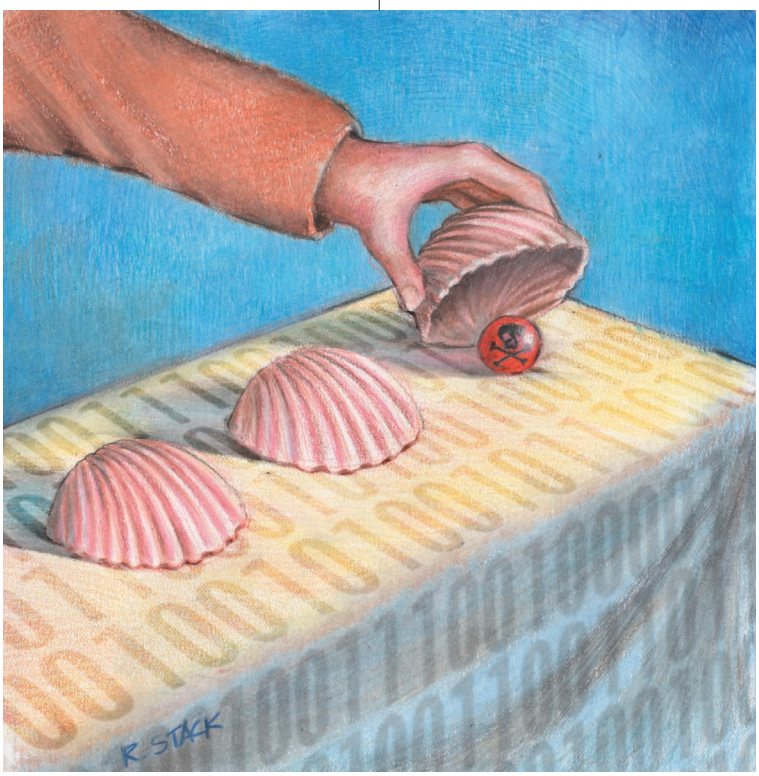
Exploit payload

Once an exploit seizes control of a vulnerable program by triggering and exploiting a bug, it immediately performs actions to achieve the exploit writer's goal. This is where the exploit payload comes in: the payload is the functional component that implements the exploit's desired purpose.

Until recently, researchers neglected exploit-payload analysis when trying to understand attackers' intent and the quality and effectiveness of their tools. Yet, the appearance of a new breed of exploit payloads, coupled with repeated failure to stop and contain automated attacks and the increased popularity of intrusion-prevention systems (IPS) that aim to detect and prevent exploitation of software bugs, has recently inspired R&D activities with a marked interest in using payload analysis to help identify future trends in the ongoing struggle to conquer the information security battlefield's high ground.

Exploit payload evolution

To better understand the payloads used in today's exploits, let's look at



some background information and historical data points to try to extrapolate the attackers' intents and assumptions.

Today the term 'shellcode' is almost synonymous for exploit payload.

Add a user account

For many years, the easiest way for an attacker to access a vulnerable system was to make the exploit modify the system's configuration to let the attacker pose as a legitimate user.

On Unix systems, this was possible by simply adding a line to the system password file (`/etc/passwd`)—exploit code from the early '90s used this approach to provide attackers direct access to compromised systems. A variation was to change the password on an existing account (typically, the privileged `root` account).

Obviously, this simplistic payload would overcome only the most basic security mechanism in today's systems and networks. Foremost, file-system integrity-checking tools such as Tripwire (www.tripwire.org), an open-source tool popularized among Unix administrators in the mid '90s, would likely alert system administrators to such changes in the operating system's password file.

Moreover, a payload of this sort couldn't guarantee access to a recently compromised system. The attacker would need to access the system through the paths available to legitimate users, and the successful exploitation of a vulnerability through one attack vector doesn't necessarily imply connectivity through legitimate means. In particular, firewalls and other filtering devices could prevent an external attacker from logging in to a vulnerable system, even after creating a valid user account on it.

Finally, use recently created user account at late or otherwise uncommon hours, and from unknown systems, is almost guaranteed to alert system administrators and security officials. Knowledgeable attackers would have to quickly cover their tracks and become as invisible as possible after running an add-a-user-account exploit.

Nonetheless, this naïve and simple payload can still be found in today's exploits—notably, some that target Microsoft operating system vulnerabilities such as the MS RPC/DCOM vulnerability used by the 2003 Blaster worm (see www.k-otik.com/exploits/09.20.rpcdcom2ver1.1.c.php, for example).

Change the system configuration

Many publicly available exploits use a simple evolution in exploit payload, altering vulnerable systems' configurations in slightly more subtle ways. In one simple variation, the payload appends a line to the Internet services daemon (`inetd`) configuration file on Unix systems (`/etc/inetd.conf`) to set up access to the compromised system by having the attacker connect to a given network port. Several exploits that target Unix vulnerabilities use a slight modification in which the exploit payload creates an alternate configuration file for `inetd` and runs it using the alternate configuration. The following excerpt from an exploit for vulnerabilities in Sun's Solaris RPC `statd` and `automount` services, originally disclosed in June 1999 (www.kb.cert.org/vuls/id/18287), illustrates this approach:

```
echo "ingreslock stream
    tcp nowait root
    /bin/sh sh -i"
    >>/tmp/bob;
    /usr/sbin/inetd -s
    /tmp/bob &"
```

A detailed account of a security incident that used this payload indicates that, although successful compromise of vulnerable systems can be achieved with a simple payload, the attacker must address several drawbacks of the exploit to remain unnoticed on the compromised system (see www.giac.org/practical/GSEC/Sara_Dearing_GSEC.pdf).

Shellcode

The next evolution in exploit payloads, known as the shellcode, became so prevalent by the mid '90s that today the term is almost synonymous for exploit payload.

The fundamental concept is that, upon seizing control of the vulnerable program by modifying its execution flow, the exploit immediately spawns a command interpreter—a *shell*, in Unix parlance—that lets the attacker interactively enter commands to be executed on the vulnerable system and read back the output. To accomplish this, the payload must have the appropriate programming instructions, or code, to run the command interpreter; hence, the name shellcode.

AlephOne's classic paper provided a clear, and very didactic, explanation of how to develop such exploits. Once the payload spawns a shell, the attacker can issue any other commands for it to interpret and execute. According to the author, several publicly available exploits were already using shellcode payloads at the time of publication (November 1996).

Shellcode's widespread adoption as the payload of choice signals an advancement in exploit quality and sophistication. Using shellcode payloads, the attacker avoids the need to alter the compromised system's file system in any way to achieve his or her goals, thus reducing the chances of being detected by the system's administrator.

However, writing the proper shellcode for any given operating system and making it work within a particular exploit requires the ex-

ploit writer to acquire and develop basic assembly and C programming skills as well as an understanding of the target operating system's foundations and APIs for spawning new processes and threads. The few fully functional examples that became publicly available for several Unix operating systems have since been used in hundreds of exploits over the past decade.

Once developers solved the technicalities of how to spawn a command interpreter in an efficient, operating-system version-independent manner, exploit writers quickly adopted the shellcode paradigm for Windows operating system vulnerabilities as well.

Network-aware shellcode

Shellcode is most suitable for exploits in which the attacker already has interactive access to the vulnerable system and just needs elevated privileges. By taking advantage of software flaws in programs running on the system on which the attacker has access to the shellcode, exploit writers avoid the complexity of managing communications between the spawn shell and any remote system the attacker might launch.

To gain access to the system, however, the attacker needs some sort of tool; exploit writers therefore adapted and further developed the shellcode payload to provide the same capabilities to an attack on a remote system. The shellcode thus becomes aware of the network topologies in which the vulnerable system and attacker are present.

The simplest network-aware shellcode spawns a command interpreter that functions as a typical network server program (or *daemon*, in Unix parlance). The shellcode listens for incoming connections on an attacker-specified network port and protocol (usually TCP, though other transport protocols are also usable); after receiving a connection from a remote system, the shellcode spawns a shell that redirects its input and output to the remote system from which the network connection was received. Presumably, the attacker will be the one connecting from the remote system and served by a fully functional command interpreter. This is typically known as a bind shellcode, or *bindshell*, because it uses the Unix sockets library's `bind` function (www.ecst.csuchico.edu/~beej/guide/net/html/syscalls.html#bind) to achieve its goal.

The presence of a firewall or any filtering device in the topological path between the attacker and the compromised system can thwart an attack. Even if the exploit works and the bindshell payload successfully creates a shell service on the target system, a firewall that blocks incoming connections to the port the attacker chooses will deny shell access to the system. To address this problem, attackers often use a variation of the bindshell, called the *reverse shell*.

The reverse shell successfully copes with firewalls and filtering devices that deny incoming connections to ports where no legitimate services are supposed to run on the protected systems. It does so by initiating the connection to the attacker from the compromised system. That is, the reverse shell spawns a shell and establishes an outgoing connection from the compromised system to the attacker's system, and the shellcode transfers control of the command interpreter to the remote system once the connection is established. However, imposing strict rules on outgoing connections or the use of proxy servers can disrupt reverse-shellcode attacks.

A subsequent advance in shellcode is the *findsocket* or *reuse-connection* shellcode, which identifies the network connection that was used to trigger and exploit the security bug and spawns a shell that uses the same connection for interactive communication with the attacker. This clever improvement of shellcode seeks to cope with network topology and firewall policy restrictions by allowing the attacker to use a communications channel known to work for exploiting the vulnerability to run the shellcode.

Multiple versions and variations of these basic network-aware shellcodes are publicly available for common operating systems, and the vast majority of exploits for recent vulnerabilities use one or more of these network-aware shellcodes. Notably, the devastating worms of 2004 and previous years used network-aware shellcode that downloads and runs multipurpose agent programs from remote Internet systems while maintaining the basic spawn-shell functionality available on some network port of the exploit writer's choosing.

Note that all variations of the network-aware shellcode rely on and assume that the payload has enough access privileges to the file system of the compromised system to load and execute a command interpreter or any other program of the exploit developer's choice.

The new shellcode

Deploying security countermeasures to mitigate the effects of exploiting software flaws in vulnerable programs can disrupt shellcode functionality and force exploit developers to revisit their code. The

Researchers and attackers are devising new types of shellcode that can avoid or bypass the restrictions these defensive technologies impose.

Please hold— Mr. Worm is on the other line

In the September/October 2003 edition of Attack Trends ("The Rise of the Gadgets," vol. 1, no. 5, pp. 78–81), I examined threats posed by the emergence of networkable devices, the most ubiquitous of which today is the cell phone.

The recent discovery of what many believe to be the first cell phone worm has proven the warning to be correct. This proof-of-concept worm infects Nokia Series 60 smart phones (which run the Symbian operating system) by copying itself to nearby devices via Bluetooth.

Receiving devices display a message asking users whether they want to accept the message from the other device. If so, the device displays a notice of a new message; when the user views it, the device warns that the attached application is "untrusted" and queries the user whether to install the worm.

If installed, the worm will execute, attempting to send itself to any Bluetooth devices it can find, every time the phone turns on. (Constant scanning for Bluetooth devices will also drain the battery at a faster rate than normal.)

While this worm requires significant user assistance to spread, many self-propagating worms successfully fool users into executing them as well. In any event, this worm represents minimal risk—it doesn't appear to have a malicious payload—but it serves to remind us that cell phones and other gadgets are vulnerable to a range of threats as they become more complex.

simplest practice to illustrate this is to run programs that provide network services, or are otherwise exposed to network attacks, with lowered privileges or with limited access to the file system.

The use of application sandboxing, IPSs, and operating-system virtualization technologies can severely hinder network-aware shellcode. As a result, researchers and attackers are devising new types of shellcode that can avoid or bypass the restrictions these defensive technologies impose. In August 2002, researchers at Core Security's labs published an alternative to the known exploit payload based on technology we coined *syscall proxying*. The proposed payload maintains the known shellcodes' network-awareness characteristics but doesn't rely on any type of access to the compromised system's file system or attempt to spawn a command interpreter; instead, it gives the attacker direct access to the vulnerable operating system's system-calls API and provides other advantages over traditional shellcodes (see www.corest.com/common/showdoc.php?idx=259&idxseccion=11).

Another payload type, the *loader payload*, mimics an operating system's initialization or boot-strapping steps. Initially, the exploit uses a very small and simple portion of code, which it loads and executes immediately after seizing control of the vulnerable program. In turn, this initial payload loads and executes code received from the attacker, whether over the same network connection or another, to implement the attack's specific goals. Two research examples of this concept are Gerardo Richarte's InlineEgg project (http://oss.coresecurity.com/projects/inlineegg.html) and Dave Aitel's MOSDEF (www.immunitysec.com/downloads/MOSDEF0.6.tgz).

Shellcode factorization and component reuse is an R&D area that emerged after the Last Stage of Delirium research group from Poland published a compendium of shellcode for various platforms (www.lsd-pl.net/documents/asmcodes-1.0.2.pdf).

We can trace another facet of the struggle between attackers and defenders back to the CanSecWest security conference in Vancouver in 2001, at which Canadian information security researcher Shane Macaulay introduced the concepts of polymorphism and code obfuscation in avoiding exploit-payload detection by network intrusion-detection systems (www.ktwo.ca/c/ADMmutate-0.8.4.tar.gz).

Some open-source projects related to exploit code and payloads have gained traction in recent years. These include exploit-development frameworks such as Metasploit (www.metasploit.org), led by security researcher H.D. Moore, and the ShellForge automatic shellcode generator developed by Phillipe Biodi and presented publicly in April 2004 (www.cansecwest.com/csw04/csw04-Biondi.pdf).

Although none of the new types of payloads I've mentioned here has yet been found in exploit code captured in the wild during or after a security compromise on live Internet systems, a marked interest in exploit-payload improvement is clearly growing. Several new research projects have emerged in the past three years, and the sophisticated payloads that have grown from these efforts will soon be typical exploits.

The authors of automated attack tools and malware that currently use network-aware shellcodes could rapidly adopt the new shellcodes and gain a definite advantage over those trying to secure their networks. To better understand and prepare our networks for future threats, we must keep alert now to advances in exploit-code development and the tools employed to create it. ▢

Iván Arce *is chief technology officer and cofounder of Core Security Technologies, based in Boston. Previously, he worked as vice president of research and development for a computer telephony integration company and as an information security consultant and software developer for various government agencies and financial and telecommunications companies. Contact him at ivan.arce@coresecurity.com.*