Shellcode

In hacking, a **shellcode** is a small piece of code used as the payload in the exploitation of a software vulnerability. It is called "shellcode" because it typically starts a command shell from which the attacker can control the compromised machine, but any piece of code that performs a similar task can be called shellcode. Because the function of a payload is not limited to merely spawning a shell, some have suggested that the name shellcode is insufficient.[1] However, attempts at replacing the term have not gained wide acceptance. Shellcode is commonly written in machine code.

Understanding shellcode and eventually writing your own is, for many reasons, an *essential skill*.[2] The most important task when creating shellcode is to make it *small* and *executable*. When creating shellcode, it must be made as small as possible so that it can be used in as wide a variety of situations as possible.[2] An interesting fact about shellcode is that it is as much an *art* as it is a science.[3] In assembly code, the same function can be performed in a multitude of ways. Some of the opcodes are shorter than others, and good shellcode writers put these small opcodes to use.[4] Some reached the smallest possible size while maintaining stability.[5]

Contents

Types of shellcode

Shellcode can either be *local* or *remote*, depending on whether it gives an attacker control over the machine it runs on (local) or over another machine through a network (remote).

Local

Local shellcode is used by an attacker who has limited access to a machine but can exploit a vulnerability, for example a buffer overflow, in a higher-privileged process on that machine. If successfully executed, the shellcode will provide the attacker access to the machine with the same higher privileges as the targeted process.

Remote

Remote shellcode is used when an attacker wants to target a vulnerable process running on another machine on a local network, intranet, or a remote network. If successfully executed, the shellcode can provide the attacker access to the target machine across the network. Remote shellcodes normally use standard TCP/IP socket connections to allow the attacker access to the shell on the target machine. Such shellcode can be categorized based on how this connection is set up: if the shellcode establishes the connection, it is called a "reverse shell" or a *connect-back* shellcode because the shellcode *connects back* to the attacker's machine. On the other hand, if the attacker establishes the connection, the shellcode is called a *bindshell* because the shellcode *binds* to a certain port on the victim's machine. There's a peculiar shellcode named *bindshell random port* that skips the binding part and listens on a random port made available by the operating system. Because of that the *bindshell random port (https://github.com/geyslan/SLAE/blob/master/improvements/tiny_shell_bind_tcp_random_port_x86_64.asm)* became the smallest and stable bindshell shellcode for x86_64 available to this date. A third, much less common type, is *socket-reuse* shellcode. This type of shellcode is sometimes used when an exploit establishes a connection to the vulnerable process that is not closed before the shellcode is run. The shellcode can then *re-use* this connection to communicate with the attacker. Socket re-using shellcode is more elaborate, since the shellcode needs to find out which connection to re-use and the machine may have many connections open.[6]

A firewall can be used to detect outgoing connections made by connect-back shellcode as well as incoming connections made by bindshells. They can therefore offer some protection against an attacker, even if the system is vulnerable, by preventing the attacker from connecting to the shell created by the shellcode. This is one reason why socket re-using shellcode is sometimes used: it does not create new connections and therefore is harder to detect and block.

Download and execute

Download and execute is a type of remote shellcode that *downloads and executes* some form of malware on the target system. This type of shellcode does not spawn a shell, but rather instructs the machine to download a certain executable file off the network, save it to disk and execute it. Nowadays, it is commonly used in drive-by download attacks, where a victim visits a malicious webpage that in turn attempts to run such a download and execute shellcode in order to install software on the victim's machine. A variation of this type of shellcode downloads and loads a library.[7][8] Advantages of this technique are that the code can be smaller, that it does not require the shellcode to spawn a new process on the target system, and that the shellcode does not need code to clean up the targeted process as this can be done by the library loaded into the process.

Staged

When the amount of data that an attacker can inject into the target process is too limited to execute useful shellcode directly, it may be possible to execute it in stages. First, a small piece of shellcode (stage 1) is executed. This code then downloads a larger piece of shellcode (stage 2) into the process's memory and executes it.

Egg-hunt

This is another form of *staged* shellcode, which is used if an attacker can inject a larger shellcode into the process but cannot determine where in the process it will end up. Small *egg-hunt* shellcode is injected into the process at a predictable location and executed. This code then searches the process's address space for the larger shellcode (the *egg*) and executes it.[9]

Omelette

This type of shellcode is similar to *egg-hunt* shellcode, but looks for multiple small blocks of data (*eggs*) and recombines them into one larger block (the *omelette*) that is subsequently executed. This is used when an attacker can only inject a number of small blocks of data into the process.[10]

Shellcode execution strategy

An exploit will commonly inject a shellcode into the target process before or at the same time as it exploits a vulnerability to gain control over the program counter. The program counter is adjusted to point to the shellcode, after which it gets executed and performs its task. Injecting the shellcode is often done by storing the shellcode in data sent over the network to the vulnerable process, by supplying it in a file that is read by the vulnerable process or through the command line or environment in the case of local exploits.

Shellcode encoding

Because most processes filter or restrict the data that can be injected, shellcode often needs to be written to allow for these restrictions. This includes making the code small, null-free or alphanumeric. Various solutions have been found to get around such restrictions, including:

- Design and implementation optimizations to decrease the size of the shellcode.
- Implementation modifications to get around limitations in the range of bytes used in the shellcode.
- Self-modifying code that modifies a number of the bytes of its own code before executing them to re-create bytes that are normally impossible to inject into the process.

Since intrusion detection can detect signatures of simple shellcodes being sent over the network, it is often encoded, made self-decrypting or polymorphic to avoid detection.

Percent encoding

Exploits that target browsers commonly encode shellcode in a JavaScript string using percent-encoding, escape sequence encoding "\uXXXX" or entity encoding.[11] Some exploits also obfuscate the encoded shellcode string further to prevent detection by IDS.

For example, on the IA-32 architecture, here's how two `NOP` (no-operation) instructions would look, first unencoded:

```
90              NOP
90              NOP
```

Encoded double-NOPs:

percent-encoding	`unescape("%u9090")`
unicode literal	`"\u9090"`
HTML/XML entity	`"&#x9090;"` or `"&#37008;"`

This instruction is used in NOP slides.

Null-free shellcode

Most shellcodes are written without the use of null bytes because they are intended to be injected into a target process through null-terminated strings. When a null-terminated string is copied, it will be copied up to and including the first null but subsequent bytes of the shellcode will not be processed. When shellcode that contains nulls is injected in this way, only part of the shellcode would be injected, making it incapable of running successfully.

To produce null-free shellcode from shellcode that contains null bytes, one can substitute machine instructions that contain zeroes with instructions that have the same effect but are free of nulls. For example, on the IA-32 architecture one could replace this instruction:

```
B8 01000000    MOV EAX,1          // Set the register EAX to 0x000000001
```

which contains zeroes as part of the literal (`1` expands to `0x00000001`) with these instructions:

```
33C0           XOR EAX,EAX        // Set the register EAX to 0x000000000
40             INC EAX            // Increase EAX to 0x00000001
```

which have the same effect but take fewer bytes to encode and are free of nulls.

Alphanumeric and printable shellcode

In certain circumstances, a target process will filter any byte from the injected shellcode that is not a printable or alphanumeric character. Under such circumstances, the range of instructions that can be used to write a shellcode becomes very limited. A solution to this problem was published by Rix in Phrack 57[12] in which he showed it was possible to turn any code into alphanumeric code. A technique often used is to create self-modifying code, because this allows the code to modify its own bytes to include bytes outside of the normally allowed range, thereby expanding the range of instructions it can use. Using this trick, a self-modifying decoder can be created that initially uses only bytes in the allowed range. The main code of the shellcode is encoded, also only using bytes in the allowed range. When the output shellcode is run, the decoder can modify its own code to be able to use any instruction it requires to function properly and then continues to decode the original shellcode. After decoding the shellcode the decoder transfers control to it, so it can be executed as normal. It has been shown that it is possible to create arbitrarily complex shellcode that looks like normal text in English.[13]

Unicode proof shellcode

Modern programs use Unicode strings to allow internationalization of text. Often, these programs will convert incoming ASCII strings to Unicode before processing them. Unicode strings encoded in UTF-16 use two bytes to encode each character (or four bytes for some special characters). When an ASCII (Latin-1 in general) string is transformed into UTF-16, a zero byte is inserted after each byte in the original string. Obscou proved in Phrack 61[14] that it is possible to write shellcode that can run successfully after this transformation. Programs that can automatically encode any shellcode into alphanumeric UTF-16-proof shellcode exist, based on the same principle of a small self-modifying decoder that decodes the original shellcode.

Platforms

Most shellcode is written in machine code because of the low level at which the vulnerability being exploited gives an attacker access to the process. Shellcode is therefore often created to target one specific combination of processor, operating system and service pack, called a platform. For some exploits, due to the constraints put on the shellcode by the target process, a very specific shellcode must be created. However, it is not impossible for one shellcode to work for multiple exploits, service packs, operating systems and even processors.[15] Such versatility is commonly achieved by creating multiple versions of the shellcode that target the various platforms and creating a header that branches to the correct version for the platform the code is running on. When executed, the code behaves differently for different platforms and executes the right part of the shellcode for the platform it is running on.

Shellcode analysis

Shellcode cannot be executed directly. In order to analyze what a shellcode attempts to do it must be loaded into another process. One common analysis technique is to write a small C program which holds the shellcode as a byte buffer, and then use a function pointer or use inline assembler to transfer execution to it. Another technique is to use an online tool, such as shellcode_2_exe, to embed the shellcode into a pre-made executable husk which can then be analyzed in a standard debugger. Specialized shellcode analysis tools also exist, such as the iDefense sclog project which was originally released in 2005 as part of the Malcode Analyst Pack. Sclog is designed to load external shellcode files and execute them within an API logging framework. Emulation based shellcode analysis tools also exist such as the sctest application which is part of the cross platform libemu package. Another emulation based shellcode analysis tool, built around the libemu library, is scdbg which includes a basic debug shell and integrated reporting features.

See also

- Alphanumeric code
- Computer security
- Buffer overflow
- Exploit (computer security)
- Heap overflow
- Metasploit Project
- Shell (computing)
- Shell shoveling
- Stack buffer overflow
- Vulnerability (computing)

References

1. Foster, James C.; and Price, Mike (April 12, 2005). *Sockets, Shellcode, Porting, & Coding:*

Reverse Engineering Exploits and Tool Coding for Security Professionals (https://books.google.com/books?id=ZNI5dvBSfZoC&printsec=frontcover&q=). Elsevier Science & Technology Books. ISBN 1-59749-005-9.

2. Chris Anley; Jack Koziol (2007). *The shellcoder's handbook: discovering and exploiting security holes* (2nd ed.). Indianapolis, IN: Wiley. ISBN 978-0-470-19882-7. OCLC 173682537 (https://www.worldcat.org/oclc/173682537).
3. Gupta, Sunil (2019), "Buffer Overflow Attack", *Ethical Hacking – Orchestrating Attacks*, Berkeley, CA: Apress, doi:10.1007/978-1-4842-4340-4_12 (https://doi.org/10.1007%2F978-1-4842-4340-4_12), ISBN 978-1-4842-4340-4
4. Foster, James C. (2005). *Buffer overflow attacks: detect, exploit, prevent*. Rockland, MA: Syngress. ISBN 1-59749-022-9. OCLC 57566682 (https://www.worldcat.org/oclc/57566682).
5. "Tiny Execve sh - Assembly Language - Linux/x86" (https://github.com/geyslan/SLAE/blob/master/4th.assignment/tiny_execve_sh.asm). *GitHub*. Retrieved 2021-02-01.
6. BHA (6 June 2013). "Shellcode/Socket-reuse" (http://www.blackhatlibrary.net/Shellcode/Socket-reuse). Retrieved 2013-06-07.
7. SkyLined (11 January 2010). "Download and LoadLibrary shellcode released" (https://web.archive.org/web/20100123014637/http://skypher.com/index.php/2010/01/11/download-and-loadlibrary-shellcode-released/). Archived from the original (http://skypher.com/index.php/2010/01/11/download-and-loadlibrary-shellcode-released/) on 23 January 2010. Retrieved 2010-01-19.
8. SkyLined (11 January 2010). "Download and LoadLibrary shellcode for x86 Windows" (https://code.google.com/p/w32-dl-loadlib-shellcode/). Retrieved 2010-01-19.
9. Skape (9 March 2004). "Safely Searching Process Virtual Address Space" (http://www.hick.org/code/skape/papers/egghunt-shellcode.pdf) (PDF). nologin. Retrieved 2009-03-19.
10. SkyLined (16 March 2009). "w32 SEH omelet shellcode" (https://web.archive.org/web/20090323030636/http://skypher.com/wiki/index.php?title=Shellcode%2Fw32_SEH_omelet_shellcode). Skypher.com. Archived from the original (http://skypher.com/wiki/index.php?title=Shellcode/w32_SEH_omelet_shellcode) on 23 March 2009. Retrieved 2009-03-19.
11. JavaScript large unescape (http://www.iss.net/security_center/reference/vuln/JavaScript_Large_Unescape.htm) IBM internet security systems
12. Rix (8 November 2001). "Writing ia32 alphanumeric shellcodes" (http://www.phrack.org/issues.html?issue=57&id=15#article). Phrack. Retrieved 2008-02-29.
13. Mason, Joshua; Small, Sam; Monrose, Fabian; MacManus, Greg (November 2009). "English Shellcode" (http://www.cs.jhu.edu/~sam/ccs243-mason.pdf) (PDF). Retrieved 2010-01-10.
14. Obscou (13 August 2003). "Building IA32 'Unicode-Proof' Shellcodes" (http://www.phrack.org/issues.html?issue=61&id=11#article). Phrack. Retrieved 2008-02-29.
15. Eugene (11 August 2001). "Architecture Spanning Shellcode" (http://www.phrack.org/issues.html?issue=57&id=14#article). Phrack. Retrieved 2008-02-29.

External links

- Shell-Storm (http://www.shell-storm.org/shellcode/) Database of shellcodes Multi-Platform.
- An introduction to buffer overflows and shellcode (http://www.phrack.org/issues.html?issue=49&id=14#article)
- The Basics of Shellcoding (http://www.infosecwriters.com/text_resources/pdf/basics_of_shellcoding.pdf) (PDF) An overview of x86 shellcoding by Angelo Rosiello (http://www.rosiello.org/)
- An introduction to shellcode development (https://web.archive.org/web/20120109070051/http://goodfellas.shellcode.com.ar/docz/bof/Writing_shellcode.html)
- Contains x86 and non-x86 shellcode samples and an online interface for automatic shellcode generation and encoding, from the Metasploit Project (https://web.archive.org/web/20080302111910/http://www.metasploit.com/shellcode/)

- a shellcode archive, sorted by Operating system (https://web.archive.org/web/20060619025456/http://www.linux-secure.com/endymion/shellcodes/).
- Microsoft Windows and Linux shellcode design tutorial going from basic to advanced (https://web.archive.org/web/20061112203748/http://www.milw0rm.com/papers/11).
- Windows and Linux shellcode tutorial containing step by step examples (http://www.vividmachines.com/shellcode/shellcode.html).
- Designing shellcode demystified (http://www.enderunix.org/docs/en/sc-en.txt)
- ALPHA3 (https://code.google.com/p/alpha3/) A shellcode encoder that can turn any shellcode into both Unicode and ASCII, uppercase and mixedcase, alphanumeric shellcode.
- Writing Small shellcode by Dafydd Stuttard (https://web.archive.org/web/20061115040739/http://www.ngssoftware.com/research/papers/WritingSmallShellcode.pdf) A whitepaper explaining how to make shellcode as small as possible by optimizing both the design and implementation.
- Writing IA32 Restricted Instruction Set Shellcode Decoder Loops by SkyLined (http://skypher.com/wiki/index.php?title=Www.edup.tudelft.nl/~bjwever/whitepaper_shellcode.html.php) A whitepaper explaining how to create shellcode when the bytes allowed in the shellcode are very restricted.
- BETA3 (https://code.google.com/p/beta3/) A tool that can encode and decode shellcode using a variety of encodings commonly used in exploits.
- Shellcode 2 Exe (http://sandsprite.com/shellcode_2_exe.php) - Online converter to embed shellcode in exe husk
- Sclog (https://github.com/dzzie/sclog) - Updated build of the iDefense sclog shellcode analysis tool (Windows)
- Libemu (https://archive.is/20130219020328/http://libemu.carnivore.it/) - emulation based shellcode analysis library (*nix/Cygwin)
- Scdbg (http://sandsprite.com/blogs/index.php?uid=7&pid=152) - shellcode debugger built around libemu emulation library (*nix/Windows)
