NEXT

TOOLS

JOHN THE RIPPER

JTR CHEAT SHEET

This cheat sheet presents tips and tricks for using JtR

JtR Community Edition - Linux

Download the JtR Bleeding Jumbo edition with improved capabilities and other goodies.

```
git clone https://github.com/magnumripper/JohnTheRipper -b bleeding-jumbo
```

Compile JtR and enable/disable required features

```
cd JohnTheRipper/
cd src/
./configure
make clean && make -s
```

```
Enable bash completion. add the following line to your ~/.bashrc
. <JtR path>/run/john.bash_completion
```

Cracking Modes

Wordlist Mode (dictionary attack)

```
./john --wordlist=password.lst hashfile
```

Mangling Rules Mode (hybrid)

```
./john --wordlist=password.lst --rules:<rulename> hashfile
```

Incremental mode (Brute Force)

```
./john --incremental hashfile
```

External mode (use a program to generate guesses)

```
./john --external: <rulename> hashfile
```

```
Loopback mode (use POT as wordlist)
./john --loopback hashfile
```

```
Mask mode (read MASK under /doc)
./john --mask=?1?1?1?1?1?1?1 -1=[A-Z] hashfile -min-len=8
```

```
Hybrid Mask mode
./john -w=password.lst -mask='?l?l?w?l?l' hashfile
```

```
Markov mode (Read MARKOV under /doc).
First generate Markov stats:
./calc_stat wordlist markovstats
Then run:
./john -markov:200 -max-len:12 hashfile --mkv-stats=markovstats
```

```
Prince mode  (Read PRINCE under /doc)
./john --prince=wordlist hashfile
```

Most modes have Maxlen=13 in John.conf but it can be overwritten with `-max-len=N` up to 24

Multiple CPU or GPU

List OpenCL devices and get the device id

```
./john --list=opencl-devices
```

List formats supported by OpenCL

```
./john --list=formats --format=opencl
```

Multiple GPU's

```
./john hashes --format:<openclformat> --wordlist:<> --rules:<> --dev=0,1 --fork=2
```

Multiple CPU's (e.g., 4 cores)

```
./john hashes --wordlist:<> --rules:<> --dev=2 --fork=4
```

Rules

```
--rules:Single
--rules:Wordlist
--rules:Extra
--rules:Jumbo  (all the above)
--rules:KoreLogic
--rules:All  (all the above)
```

Incremental Modes (Brute Force)

```
--incremental:Lower (26 char)
--incremental:Alpha (52 char)
--incremental:Digits (10 char)
--incremental:Alnum (62 char)
```

Incremental mode with new charsets

Create a new charset based on john.pot

```
./john --make-charset=charset.chr
```

Create a new entry in John.conf to accommodate the new charset

```
# Incremental modes
[Incremental:charset]
File = $JOHN/charset.chr
MinLen = 0
MaxLen = 31
CharCount = 95
```

Run JtR with the new charset

```
./john --incremental=charset hashfile
```

Wordlists

Sort a wordlist to use with wordlist rule mode

```
$tr A-Z a-z < SOURCE | sort -u > TARGET
```

Use a POT file to generate a new wordlist

```
cut -d: -f2 john.pot | sort -u > pot.dic
```

Generate candidate passwords for slow hashes.

```
./john --wordlist= password.lst --stdout --rules:Jumbo | ./unique -mem=25 wordlist.uniq
```

Use external mode for complex rules

http://www.lanmaster53.com/2011/02/creating-complex-password-lists-with-john-the-ripper/

Generate a wordlist that meets the complexity specified in the complex filter

./john --wordlist=[path to word list] --stdout --external:[filter name] > [path to output list]

Try sequences of adjacent keys on a keyboard as candidate passwords

```
john --external:Keyboard hashfile
```

Configuration Items on John.conf

When using both CPU and GPU set this flag

Idle = N

Hidden Options

```
./john --list=hidden-options
```

Display guesses

```
./john --incremental:Alpha -stdout -session=s1
```

Generate guesses with external program

```
crunch 1 6 abcdefg | ./john hashes -stdin -session=s1
```

Session and Restore

```
./john hashes -session=name
```

./john --restore:name

Show cracked passwords

```
./john hashes --pot=<>  --show
```

Resources

John-Users Mailing List

http://www.openwall.com/lists/john-users/

JtR Community Wiki
http://openwall.info/wiki/john

`Documentation under doc folder`

Matt Weir Blog
http://reusablesec.blogspot.ch/

Simple Rule in John.conf

```
[List.Rules:Tryout]
l
u
c
l r
l Az"2015"
d
l A0"2015"
A0"#"Az"#"
```

Details

convert to lowercase
`l`

convert to uppercase
`u`

#capitalize
`c`

#lowercase the word and reverse it (palindrome)
`l r`

#lowercase the word and append at end of the word (Az) the number 2015
`l Az"2015"`

duplicate
`d`

lowercase the word and prepend at beggining of the word (A0) the number 2015
`l A0"2015"`

Add # to the beginning and end of the word
`A0"#"Az"#"`

Use the Wordlist Rule

Display the password candidates generated with the mangling rule
`./john --wordlist=password.lst --stdout --rules:Tryout`

Generate password candidates max length of 8
`./john --wordlist=password.lst --stdout=8 --rules:Tryout`

`./john hashes --wordlist=password.lst --rules:Tryout`

Simple Wordlist Rules

#lowercase the first character, and uppercase the rest
`c`

#toggle case of all characters in the word
`t`

#toggle case of the character in position N
`TN`

#reverse: "Fred" -> "derF"
`r`

#duplicate: "Fred" -> "FredFred"
`d`

#reflect: "Fred" -> "FredderF"
`f`

#rotate the word left: "jsmith" -> "smithj"
`{`

#rotate the word right: "smithj" -> "jsmith"
`}`

#append character X to the word
`$X`

#prefix the word with character X
`^X`

Insert and Delete Wordlist Rules

#Remove the first char from the word
`[`

#Remove the last char from the word
`]`

#delete the character in position N
`DN`

#extract substring from position N for up to M characters
`xNM`

#insert character X in position N and shift the rest right
`iNX`

#overstrike character in position N with character X
`oNX`

Charset and Conversion Wordlist Rules

#shift case: "Crack96" -> "cRACK(^"
`S`

#lowercase vowels, uppercase consonants: "Crack96" -> "CRaCK96"
`V`

#shift each character right, by keyboard: "Crack96" -> "Vtsvl07"
`R`

#shift each character left, by keyboard: "Crack96" -> "Xeaxj85"
`L`

Length control

#reject the word unless it is less than N characters long
`<N`

#reject the word unless it is greater than N characters long
`>N`

#truncate the word at length N
`'N`

Dictionaries

Generate wordlists from Wikipedia pages: `wget https://raw.githubusercontent.com/zombiesam/wikigen/master/wwg.py`

```
python wwg.py -u http://pt.wikipedia.org/wiki/Fernando_Pessoa -t 5 -o fernandopessoa -m3
```

Generate wordlists from Aspell Dict's

```
aspell dump dicts
sudo apt-get install aspell-es
aspell -d es dump master | aspell -l es expand | awk 1 RS=" |\n" > Spanish.dic
```

Resources

Full Rules Documentation
`http://www.openwall.com/john/doc/RULES.shtml`

Password Analysis and Cracking Kit
`https://thesprawl.org/projects/pack/`

Mangling Rules Generation by Simon Marechal
`http://www.openwall.com/presentations/Passwords12-Mangling-Rules-Generation/`

SQLMAP

sqlmap Cheat Sheet

Cheat Sheet Series — comparitech

Basic options

The sqlmap command will not run without at least one of these options added to it.

-u URL	The target URL Format: -u "http://www.target.com/path/file.htm?variable=1"
-d DIRECT	Connection string for direct database connection Format: -d DBMS://DATABASE_FILEPATH or -d DBMS://USER:PASSWORD@DBMS_IP:DBMS_PORT/DATABASE_NAME
-l LOGFILE	Parse target(s) from Burp or WebScarab proxy log file
-m BULKFILE	Scan multiple targets given in a textual file Format: The file should contain a URL per line
-r REQUESTFILE	Load HTTP request from a file Format: The file can contain an HTTP request or an HTTPS transaction
-g GOOGLEDORK	Process Google dork results as target URLs
-c CONFIGFILE	Load options from a configuration INI file
--wizard	A guided execution service
--update	Update sqlmap to the latest version
--purge	Clear out the sqlmap data folder
--purge-output	As above
--dependencies	Check for missing sqlmap dependencies
-h	Basic help
-hh	Advanced help
--version	Show the sqlmap version number
-v VERBOSE	Verbosity level

Verbosity option values

Possible verbosity level values are:

0	Only Python tracebacks, error, and critical messages
1	Feedback of 0 plus information and warning messages
2	Feedback of 1 plus debug messages
3	Feedback of 2 plus the payloads injected
4	Feedback of 3 plus HTTP requests
5	Feedback of 4 plus the HTTP headers of responses
6	Feedback of 5 plus the content of the HTTP responses

Optimization

The following options can be used to improve the performance of sqlmap.

-o	Turn on all optimization switches
--predict-output	Predict common queries output
--keep-alive	Use persistent HTTP(s) connections
--null-connection	Retrieve page length without actual HTTP response body
--threads=THREADS	Max number of concurrent HTTP(s) requests (default 1)

Detection

The following options are used during research in the detection phase.

--level=LEVEL	The level of tests to perform (1-5, default 1)
--risk=RISK	The risk of tests to perform (1-3, default 1)
--string=STRING	A string to match when query is evaluated to True
--not-string=FALSE-STRING	A string to match when query is evaluated to False
--regexp=REGEXP	Regexp to match when query is evaluated to True
--code=CODE	HTTP code to match when query is evaluated to True
--smart	Perform thorough tests only if positive heuristic(s)

Brute force

These options implement checks during the launch of a brute force attack.

--common-tables	Check the existence of common tables
--common-columns	Check the existence of common columns
--common-files	Check the existence of common files

Miscellaneous

These options do not fit into any of the above categories.

-z MNEMONICS	Use short mnemonics (e.g. "flu,bat,ban,tec=EU")
--alert=ALERT	Run host OS command(s) when SQL injection is found
--beep	Beep on the question and/or when SQLi/XSS/FI is found
--disable-coloring	Disable console output coloring
--list-tampers	Display list of available tamper scripts
--offline	Work in offline mode (only use session data)
--results-file=RESULTS-FILE	Location of CSV results file in multiple targets mode
--shell	Prompt for an interactive sqlmap shell
--tmp-dir=TMPDIR	Local directory for storing temporary files
--unstable	Adjust options for unstable connections

Level option values

This option dictates the volume of tests to perform and the extent of the feedback that they will provide. A higher value implements more extensive checks.

1	A limited number of tests/requests; GET AND POST parameters will be tested (default)
2	Test cookies
3	Test cookies plus User-Agent/Referer
4	As above plus null values in parameters and other bugs
5	An extensive list of tests with an input file for payloads and boundaries

Techniques

These options relate to specific attack strategies. They adjust and focus the attack on particular techniques and targets.

--technique=TECHNIQUE	The SQL injection techniques to use (default "BEUSTQ")
--time-sec=TIMESEC	The number of seconds to delay the DBMS response (default 5)
--union-cols=UCOLS	A range of columns to test for UNION query SQL injection
--union-char=UCHAR	A character to use for brute-forcing columns
--union-from=UFROM	The table to use in the FROM part of a UNION query SQL injection
--dns-domain=DNS-DOMAIN	The domain name to use in a DNS exfiltration attack
--second-url=SECOND-URL	Resulting page URL searched for a second-order response
--second-req=SECOND-REQ	Load a second-order HTTP request from the file
-f	Perform an extensive DBMS version fingerprint
--fingerprint	As above

Request

Add these options to a command to specify how to connect to the target URL.

-A AGENT	HTTP User-Agent header value
--user-agent=AGENT	As above
-H HEADER	Extra header (e.g. "X-Forwarded-For: 127.0.0.1")
--headers=HEADERS	As above
--method=METHOD	Specify an HTTP method to use, such as POST or PUT
--data=DATA	Data string to be sent through POST (e.g. "id=1")
--param-del=PARAMETER	A character to be used for splitting parameter values (e.g., &)
--cookie=COOKIE	HTTP Cookie header value (e.g. "PHPSESSID=a8d127e..")
--cookie-del=COOKIE-CHAR	A character to be used for splitting cookie values (e.g. ;)
--live-cookies=LIVE-COOKIES	A file containing live cookies to be used for loading values
--load-cookies=LOAD-COOKIES	As above with cookies in Netscape/wget format
--drop-set-cookie	Ignore the Set-Cookie header in the response
--mobile	Imitate a smartphone through HTTP User-Agent header
--random-agent	Use a randomly selected HTTP User-Agent header value
--host=HOST	An HTTP Host header value
--referer=REFERER	An HTTP Referer header value
--auth-type=AUTH-TYPE	An HTTP authentication type (Basic, Digest, NTLM or PKI)
--auth-cred=AUTH-CRED	HTTP authentication credentials (name:password)
--auth-file=AUTH-FILE	HTTP authentication PEM cert/private key file
--ignore-code=IGNORE-CODE	Ignore (problematic) HTTP error code (e.g. 401)
--ignore-proxy	Ignore system default proxy settings
--ignore-redirects	Ignore redirection attempts
--ignore-timeouts	Ignore connection timeouts
--proxy=PROXY	Use a proxy to connect to the target URL
--proxy-cred=PROXY-LOGIN	Proxy authentication credentials (name: password)
--proxy-file=PROXY-LIST	Load proxy list from a file
--proxy-freq=PROXY-RATE	Number of requests between the change of proxy from a given list
--tor	Use Tor anonymity network
--tor-port=TORPORT	Set the Tor proxy port to be other than the default
--tor-type=TORTYPE	Set the Tor proxy type (HTTP, SOCKS4 or SOCKS5 (default))
--check-tor	Check to see if Tor is used properly
--delay=DELAY	Delay in seconds between each HTTP request
--timeout=TIMEOUT	Seconds to wait before timeout connection (default 30)
--retries=RETRIES	Number of retries upon timeout (default 3)
--randomize=RPARAM	Randomly change the value for a given parameter(s)
--safe-url=SAFEURL	URL address to visit frequently during testing
--safe-post=SAFE-POST	POST data to send to a safe URL
--safe-req=SAFE-REQUEST	Load safe HTTP request from a file
--safe-freq=SAFE-FREQ	The number of regular requests between visits to a safe URL
--skip-urlencode	Skip URL encoding of payload data
--csrf-token=CSRF-TOKEN	Parameter used to hold the anti-CSRF token
--csrf-url=CSRF-URL	URL to visit for extraction of anti-CSRF token
--csrf-method=CSRF-METHOD	HTTP method to use during anti-CSRF token page visit
--csrf-retries=CSRF-RETRIES	Number of retries to get the anti-CSRF token (default 0)
--force-ssl	Force usage of SSL/HTTPS
--chunked	Use HTTP chunked transfer encoded (POST) requests
--hpp	Use HTTP parameter pollution method
--eval=EVALCODE	Evaluate the provided Python code before the request (e.g. "import hashlib;id2=hashlib.md5(id).hexdigest()")

Injection

The following options can be used to specify which parameters to test for, provide custom injection payloads and optional tampering scripts.

-p TESTPARAMETER	Testable parameter(s)
--skip=SKIP	Skip testing for given parameter(s)
--skip-static	Skip testing parameters that do not appear to be dynamic
--param-exclude=PARAM-EXCLUDE	Regexp to exclude parameters from testing (e.g. "ses")
--param-filter=PARAM-FILTER	Select testable parameter(s) by place (e.g. "POST")
--dbms=DBMS	Force back-end DBMS to provided value
--dbms-cred=DBMS-CREDENTIALS	DBMS authentication credentials (user:password)
--os=OS	Force back-end DBMS operating system to the provided value
--invalid-bignum	Use big numbers for invalidating values
--invalid-logical	Use logical operations for invalidating values
--invalid-string	Use random strings for invalidating values
--no-cast	Turn off payload casting mechanism
--no-escape	Turn off string escaping mechanism
--prefix=PREFIX	Injection payload prefix string
--suffix=SUFFIX	Injection payload suffix string
--tamper=TAMPER	Use given script(s) for tampering injection data

Risk option values

The number given as a parameter to the risk option specifies the extent to which the actions of the tests will expose the attacker. Tests performed in the lowest level will be hardly noticeable to the user, while tests in the higher category can result in mass changes to data.

1	Quick, unnoticeable tests (default)
2	Tests that involve lengthy, heavy data processing, such as time-based SQLi
3	Adds OR-based SQLi and possible data manipulation

Operating system access

These options can be used to access the operating system supporting the DBMS.

--os-cmd=OSCMD	Execute an operating system command
--os-shell	Prompt for an interactive operating system shell
--os-pwn	Prompt for an OOB shell, Meterpreter or VNC
--os-smbrelay	One-click prompt for an OOB shell, Meterpreter or VNC
--os-bof	Stored procedure buffer overflow exploitation
--priv-esc	Database process user privilege escalation
--msf-path=MSFPATH	Local path where Metasploit Framework is installed
--tmp-path=TMPPATH	Remote absolute path of temporary files directory

General

These options provide the opportunity to set general operating parameters.

-s SESSIONFILE	Load session from a stored (.sqlite) file
-t TRAFFICFILE	Log all HTTP traffic into a text file
--answers=ANSWERS	Set predefined answers (e.g. "quit=N,follow=N")
--base64=BASE64PARAMS	Parameter(s) containing Base64 encoded data
--base64-safe	Use URL and filename safe Base64 alphabet (RFC 4648)
--batch	Never ask for user input; use the default behavior
--binary-fields=BINARY-FIELDS	The result fields in binary format (e.g., "digest")
--check-internet	Check the Internet connection before assessing the target
--cleanup	Clean up sqlmap-specific UDF and tables from the database
--crawl=CRAWLDEPTH	Crawl the website starting from the target URL
--crawl-exclude=CRAWL-EXCLUDE	Regexp to exclude pages from crawling (e.g. "logout")
--csv-del=CSVDEL	The delimiter to use in CSV output (default ",")
--charset=CHARSET	Blind SQL injection charset (e.g. "0123456789abcdef")
--dump-format=DUMP-FORMAT	The format of the data dump (CSV (default), HTML or SQLITE)
--encoding=ENCODING	Character encoding to use for data retrieval (e.g., GBK)
--eta	Display the estimated time of arrival for each output
--flush-session	Flush session files for the current target
--forms	Parse and test forms on the target URL
--fresh-queries	Ignore query results stored in the session file
--gpage=GOOGLEPAGE	Use Google dork results starting from the given page number
--har=HARFILE	Log all HTTP traffic into a HAR file
--hex	Use hex conversion during data retrieval
--output-dir=OUTPUT-DIR	The custom output directory path
--parse-errors	Parse and display DBMS error messages from responses
--preprocess=PREPROCESS	Use the named script(s) for preprocessing (request)
--postprocess=POSTPROCESS	Use the named script(s) for postprocessing (response)
--repair	Redump entries having an unknown character marker (?)
--save=SAVECONFIG	Save options to a configuration INI file
--scope=SCOPE	Regexp for filtering targets
--skip-heuristics	Skip heuristic detection of SQLi/XSS vulnerabilities
--skip-waf	Skip heuristic detection of WAF/IPS protection
--table-prefix=TABLE-PREFIX	The prefix to use for temporary tables (default: "sqlmap")
--test-filter=TEST-FILTER	Select tests by payloads and titles (e.g. ROW)
--test-skip=TEST-SKIP	Skip tests by payloads and titles (e.g., BENCHMARK)
--web-root=WEBROOT	The Web server document root directory (e.g. "/var/www")

Running an SQL injection attack scan with sqlmap

The large number of options available for sqlmap is daunting. There are too many options to comb through in order to work out how to form an SQL injection attack. The best way to acquire the knowledge of how to perform the different types of attacks is to **learn by example**.

To experience how a sqlmap test system proceeds, try the following test run, substituting the URL of your site for the marker <URL>. You need to include the schema on the front of the URL (http or https).

```
$ sqlmap.py -u "<URL>" --batch --banner
```

This command will trigger a run-through of all of the sqlmap procedures, offering you options over the test as it proceeds.

The system will show **the start time** of the test. Each report line includes the time that each test completed.

The sqlmap service will **test the connection** to the Web server and then scan various aspects of the site. These attributes include the site's default character set, a check for the presence of **defense systems**, such as a Web application firewall or intrusion detection systems.

The next phase of the test identifies the DBMS used for the site. It will attempt **a series of attacks** to probe the vulnerability of the site's database. These are:

- A GET input attack – this identifies the susceptibility to Classic SQLI and XSS attacks
- DBMS-specific attacks
- Boolean-based blind SQLI
- The system will ask for a level and a risk value. If these are high enough, it will run a time-based blind SQLI
- An error-based SQLI attack
- A UNION-based SQLI if the level and risk values are high enough
- Stacked queries

In answer to the banner option used in this run, sqlmap completes its run by fetching **the database banner**. Finally, all extracted data with explanations of their meanings are written to **a log file**.

As you can see, without many options given on the command, the sqlmap system will run

As you can see, without many options given on the command, the sqlmap system will run through a standard series of attacks and will check with the user for decisions over the depth of the test as the test progresses.

A small change in the command will run the same battery of tests but by using a **POST** as a test method instead of a **GET**.

Try the following command:

```
$ sqlmap.py -u "<URL>" --data="id=1" --banner
```

Password cracking with sqlmap

A change of just one word in the first command used for the previous section will give you a range of tests to see whether the **credentials management system** of your database has weaknesses.

Enter the following command:

```
$ sqlmap.py -u "<URL>" --batch --password
```

Again, you need to substitute your site's URL for the <URL> marker.

When you run this command, sqlmap will initiate a series of tests and give you a number of options along the way.

The sqlmap run will try a time-based blind SQLI and then a UNION-based blind attack. It will then give you the option to store password hashes to a file for analysis with another tool and then gives the opportunity for a dictionary-based attack.

The services will try a series of well-known user account names and cycle through a list of often-used passwords against each candidate username. This is called a "**cluster bomb**" attack. The files suite of sqlmap includes a file of payloads for this attack but you can supply your own file instead.

Whenever sqlmap hits a username and password combination, it will display it. All actions for the run are then written to a log file before the program ends its run.

Get a list of databases on your system and their tables

Information is power and hackers first need to know what database instances you have on

Whenever sqlmap hits a username and password combination, it will display it. All actions for the run are then written to a log file before the program ends its run.

Get a list of databases on your system and their tables

Information is power and hackers first need to know what database instances you have on your system in order to hack into them. You can find out whether this basic information can be easily accessed by **intruders** with the following command:

```
$ sqlmap.py -u "<URL>" --batch --dbs
```

This test will include time-based, error-based, and UNION-based SQL injection attacks. It will then identify the DBMS brand and then list the database names. The information derived during the test run is then written to a log file as the program terminates.

Investigate a little further and get a list of the tables in one of those databases with the following command.

```
$ sqlmap.py -u "<URL>" --batch --tables -D <DATABASE>
```

Enter the name of one of the database instances that you got from the list in the first query of this section.

This test batch includes time-based, error-based, and UNION-based SQL injection attacks. It will then list the names of the tables that are in the specified database instance. This data is written to a log file as the program finishes.

Get **the contents** of one of those tables with the following command:

```
$ sqlmap.py -u "<URL>" --batch --dump -T <TABLE> -D <DATABASE>
```

Substitute the name of one of the tables you discovered for the <TABLE> marker in that command format.

The test will perform a UNION-based SQL injection attack and then query the named table, showing its records on the screen. This information is written to a log file and then the program terminates.

Explore the Cheat Sheet

The commands shown in this guide are just the start. Su tests will

What is risk? Risk is the potential impact that a vulnerability, threat, or asset presents to an organization calculated against all other vulnerabilities, threats, and assets. Evaluating risk helps to determine the likelihood of a specific issue causing a data breach that will cause harm to an organization's finances, reputation, or regulatory compliance. Reducing risk is critical for many organizations. There are many certifications, regulatory standards, and frameworks that are designed to help companies understand, identify, and reduce risks.

- **Zero-day** – A zero-day attack is an exploit that is unknown to the world, including the vendor of the product, which means it is unpatched by the vendor. These attacks are commonly used in nation-state attacks, as well as by large criminal organizations. The discovery of a zero-day exploit can be very valuable to ethical hackers and penetration testers, and can earn them a bug bounty. These bounties are fees paid by vendors to security researchers that discover unknown vulnerabilities in their applications.

All organizations have assets that need to be kept safe; an organization's systems, networks, and assets always contain some sort of security weakness that can be taken advantage of by a hacker. Next, we'll dive into understanding what a vulnerability is.

- **Vulnerability** – A vulnerability is a weakness or security flaw that exists within technical, physical, or human systems that hackers can exploit in order to gain unauthorized access or control over systems within a network. Common vulnerabilities that exist within organizations include human error (the greatest of vulnerabilities on a global scale), misconfiguration of devices, using weak user credentials, poor programming practices, unpatched operating systems and outdated applications on host systems, using default configurations on systems, and so on.

A threat actor will look for the *lowest-hanging fruits* such as the vulnerabilities that are the easiest to be taken advantage of. The same concept applies to penetration testing. During an engagement, the penetration tester will use various techniques and tools to discover vulnerabilities and will attempt to exploit the easy ones before moving to the more complex security flaws on a target system.

- **Exploit** – An exploit is the thing, tool, or code that is used to take advantage of a vulnerability on a system. For example, take a hammer, a piece of wood, and a nail. The vulnerability is the soft, permeable nature of wood, and the exploit is the act of hammering the nail into the wood. Once a vulnerability is found on a system, the threat actor or penetration tester will either develop or search for an exploit that is able to take advantage of the security weakness. It's important to understand that the exploit should be tested on a system to ensure it has the potential to be successful when launched by the threat actor. Sometimes, an exploit may work on a system and may not work on another. Hence, seasoned penetration testers will ensure their exploits are tested and graded on their rate of success per vulnerability.
- **Risk** – While it may seem like penetration testers are hired to simulate real-world cyber-attacks on a target organization, the goal of such engagements is much deeper than it seems. At the end of the penetration test, the cybersecurity professional will present all the vulnerabilities and possible solutions to help the organization mitigate and reduce the risk of a potential cyber-attack.

Pre-engagement

During the pre-engagement phase, key personnel are selected. These individuals are key to providing information, coordinating resources, and helping the penetration testers to understand the scope, breadth, and rules of engagement in the assessment.

This phase also covers legal requirements, which typically include a **Non-Disclosure Agreement** (**NDA**) and a **Consulting Services Agreement** (**CSA**). The following is a typical process overview of what is required prior to the actual penetration testing:

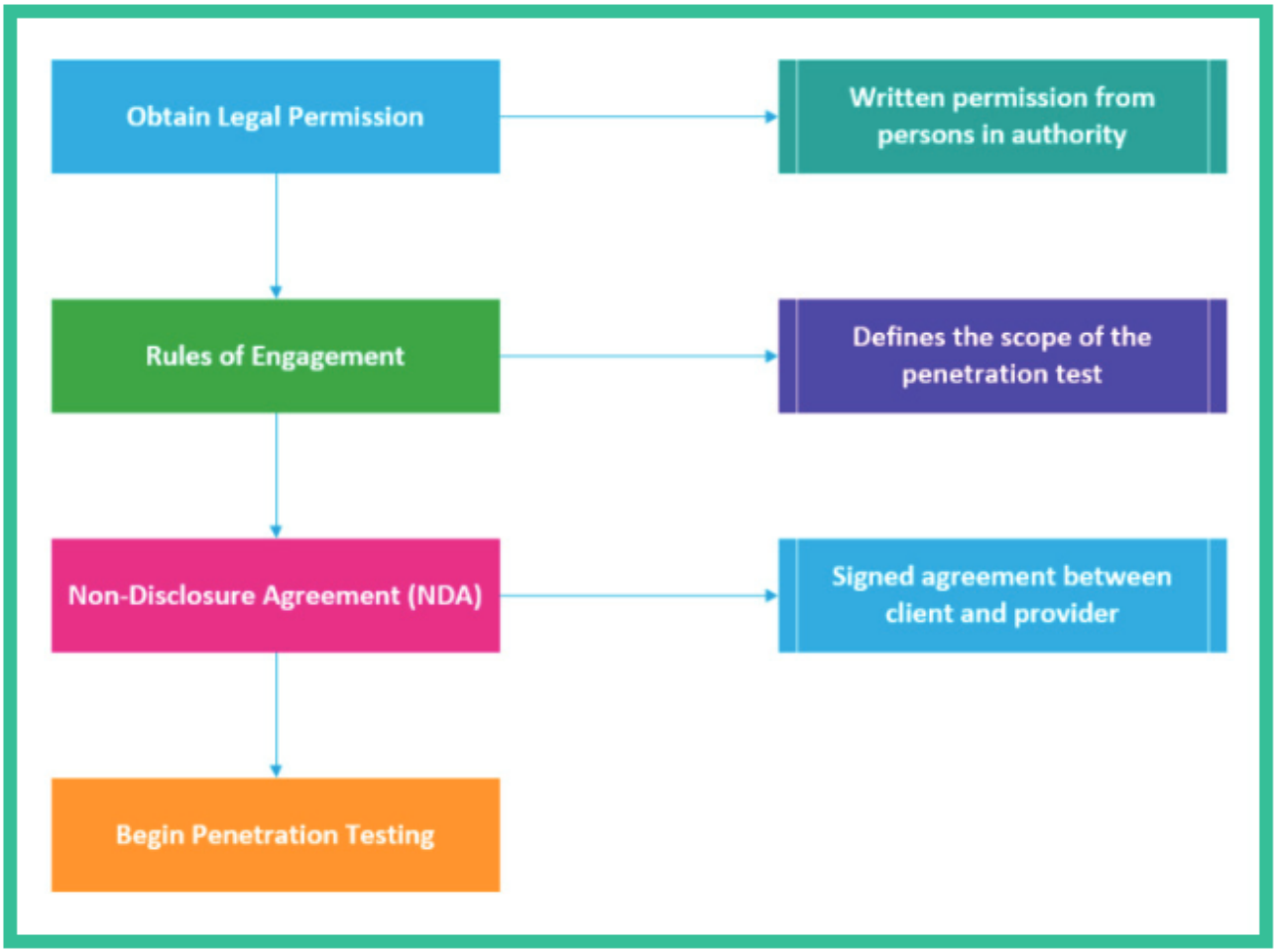


Figure 1.2 – Pre-engagement

An NDA is a legal agreement that specifies that a penetration tester and their employer will not share or hold onto any sensitive or proprietary information that is encountered during the assessment. Companies usually sign these agreements with cybersecurity companies who will, in turn, sign them with employees working on the project. In some cases, companies sign these agreements directly with the penetration testers from the company carrying out the project.

The scope of a penetration test, also known as the **rules of engagement**, defines the systems the penetration tester can and cannot hack. This ensures the penetration tester remains within legal boundaries. This is a mutual agreement between the client (organization) and the penetration tester and their employer. It also defines sensitive systems and their IP addresses as well as testing times and which systems require special testing windows. It's incredibly important for penetration testers to pay close attention to the scope of a penetration test and where they are testing in order to always stay within the testing constraints.

The following are some sample pre-engagement questions to help you define the scope of a penetration test:

- What is the size/class of your external network? (Network penetration testing)
- What is the size/class of your internal network? (Network penetration testing)
- What is the purpose and goal of the penetration test? (Applicable to any form of penetration testing)
- How many pages does the web application have? (Web application penetration testing)
- How many user inputs or forms does the web application have?

This is not an extensive list of pre-engagement questions, and all engagements should be given thorough thought to ensure that you ask all the important questions so you don't underscope or underprice the engagement.

Now that we've understood the legal limitation stages of penetration testing, let's move on to learn about the information gathering phase and its importance.

Information gathering

Penetration testing involves information gathering, which is vital to ensure that penetration testers have access to key information that will assist them in conducting their assessment. Seasoned professionals normally spend a day or two conducting extensive reconnaissance on their target. The more knowledge that is known about the target will help the penetration tester to identify the attack surface such as points of entry in the target's systems and networks. Additionally, this phase also helps the penetration tester to identify the employees, infrastructure, geolocation for physical access, network details, servers, and other valuable information about the target organization.

Understanding the target is very important before any sort of attack as a penetration tester, as it helps in creating a profile of the potential target. Recovering user credentials/login accounts in this phase, for instance, will be vital to later phases of penetration testing as it will help us gain access to vulnerable systems and networks. Next, we will discuss the essentials of threat modeling.

Threat modeling

Threat modeling is a process used to assist penetration testers and network security defenders to better understand the threats that inspired the assessment or the threats that the application or network is most prone to. This data is then used to help penetration testers simulate, assess, and address the most common threats that the organization, network, or application faces.

The following are some threat modeling frameworks:

- **Spoofing, Tampering, Repudiation, Information disclosure, Denial of server and Elevation of privilege (STRIDE)**
- **Process for Attack Simulation and Threat Analysis (PASTA)**

Having understood the threats an organization faces, the next step is to perform a vulnerability assessment on the assets to further determine the risk rating and severity.

Vulnerability analysis

Vulnerability analysis typically involves the assessors or penetration testers running vulnerability or network/port scans to better understand which services are on the network or the applications running on a system and whether there are any vulnerabilities in any systems included in the scope of the assessment. This process often includes manual vulnerability discovery and testing, which is often the most accurate form of vulnerability analysis or vulnerability assessment.

There are many tools, both free and paid, to assist us in quickly identifying vulnerabilities on a target system or network. After discovering the security weaknesses, the next phase is to attempt exploitation.

Scanning and enumeration

The second phase of hacking is scanning. Scanning involves using a direct approach in engaging the target to obtain information that is not accessible via the reconnaissance phase. This phase involves profiling the target organization, its systems, and network infrastructure.

The following are techniques used in the scanning phase:

- Checking for any live systems
- Checking for firewalls and their rules
- Checking for open network ports
- Checking for running services
- Checking for security vulnerabilities
- Creating a network topology of the target network

1. Open a terminal window and generate a GPG key

The first thing to do is open the terminal window from your desktop menu. Once it's open, you'll want to generate a GPG key with the command:

```
gpg --gen-key
```

You'll be asked to enter your real name and an email address, then type "O "to Okay the information. After that, you type/verify a passphrase for the key.

2. Change into the directory housing the file

With your key created, navigate to the folder housing the file to be encrypted. Let's say the file is in ~/Documents. Change to that directory with the command:

```
cd ~/Documents
```

3. Encrypt the file

We're going to use the *gpg* command to encrypt the file. For example, we'll encrypt the file zdnet_test with the command:

```
gpg -c zdnet_test
```

The -c option tells gpg the zdnet_test file is to be encrypted. You will then be asked to type and verify a password for the encrypted file.

Once you've encrypted the file, you'll notice there are two files: zdnet_test and zdnet_test.gpg. The file with the .gpg extension is the encrypted file. At this point, you can remove the initial test file with the command:

```
rm zdnet_test
```

4. Configure the password cache agent

Oddly enough, the GPG tool caches passwords. Because of this, you (or anyone who has access to your system) could decrypt the file without having to type the password with the command *gpg zdnet_test*. That's not safe. To get around this, we have to disable password caching for the GPG agent. To do this, create a new file with the command:

```
nano ~/.gnupg/gpg-agent.conf
```

In that file, paste the following lines:

```
default-cache-ttl 1
max-cache-ttl 1
```

Next, restart the agent with the command:

```
echo RELOADAGENT | gpg-connect-agent
```

Now, when you (or anyone) types the decrypt command, *gpg zdnet_test*, the password prompt will appear. Until that password is successfully entered, the contents of the file will remain encrypted.

The GUI (Graphical User Interface) method of encrypting files

This method is significantly more efficient.

1. Install the required software

Before you use the GUI method, make sure to take care of Steps 1 and 4 above. You only have to do this once. After that, you'll need to install a piece of software with the command:

```
sudo apt-get install seahorse-nautilus -y
```

If you're using a distribution based on RHEL or Fedora Linux, that command would be:

```
sudo dnf install seahorse-nautilus -y
```

Once installed, restart Nautilus with the command:

```
nautilus -q
```

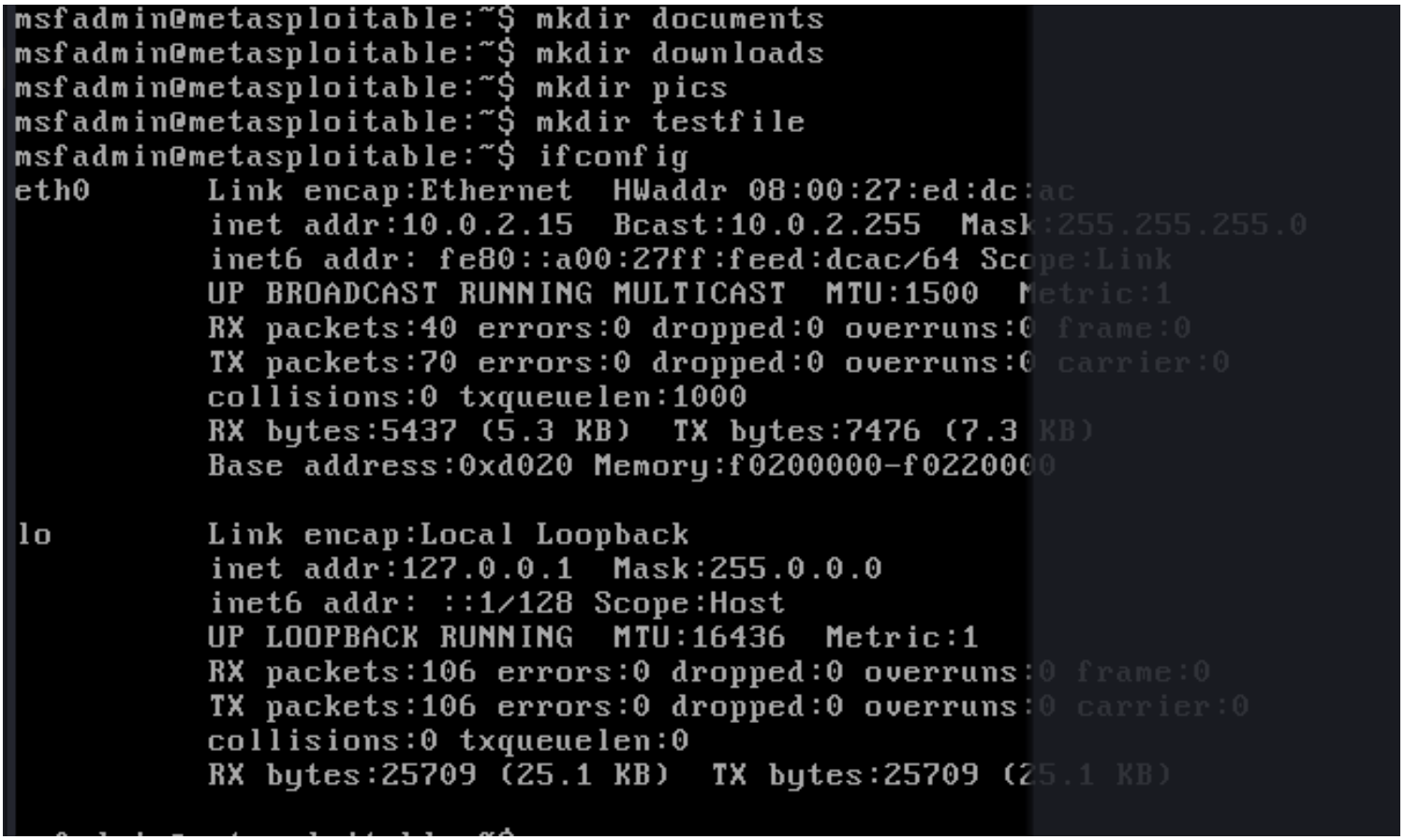
toolgoddesstt toolgoddesstt.

pro statement

I am currently dedicated to learning anything and eveything I can about linux, and penetration testing. If I dont understand something, I push forward till I am 100 percent crystal clear
I am allreaday ready for IT imtermediate level work, but cybersecurity is what interests me.. My strengths are I am persistent, stubborn smart and I love computers.
A weakness is I am stubborn and do not have a lot of extra time, but I am making it. I also lack patience, which is a everyday struggle.

I have alwahys been acutely aware of regulations and have a strong professional background in ethics. I am an actiong store manager, and I have to make ethical choices daily. There is no grey area in this, there is right and wrong. My Ethics are strong, my intelligence is right beside it.
I will continue to work on this professional statement as my life evolves, but my love for computers , experience as a leader, will change and grow over time. I will be phenomenal at my job

metasploitable 2 ip =



that I will cover in this section are the `.tar` , `.gz` , `.bz2` , and `.zip` extensions. Here's the list of commands to compress and extract different types of archives:

Tar Archive

To compress using tar extension:

```
$tar cf compressed.tar files
```

To extract a tar compressed file:

```
$tar xf compressed.tar
```

Gz Archive

To create compressed.tar.gz from files:

```
$tar cfz compressed.tar.gz files
```


To extract compressed.tar.gz:

```
$tar xfz compressed.tar.gz
```

To create a compressed.txt.gz file:

```
$gzip file.txt> compressed.txt.gz
```

To extract compressed.txt.gz:

```
$gzip -d compressed.txt.gz
```

Let's extract the `rockyou.txt.gz` file that comes initially compressed in Kali:

```
root@kali:~# gzip -d /usr/share/wordlists/rockyou.txt.gz
```

Let's extract the `rockyou.txt.gz` file that comes initially compressed in Kali:

```
root@kali:~# gzip -d /usr/share/wordlists/rockyou.txt.gz
```

Bz2 Archive

To create compressed.tar.bz2 from files:

```
$tar cfj compressed.tar.bz2 files
```

To extract compressed.tar.bz2:

```
$tar xfj compressed.tar.bz2
```

Zip Archive

To create compressed.zip from files:

```
$zip compressed.zip files
```

To extract compressed.zip files:

```
$unzip compressed.zip
```

Manipulating Directories in Kali

To print the current working directory, you must use the `pwd` command to get the job done (don't mix up the `pwd` command with `passwd` command; they're two different things):

```
$pwd
```

To change the current working directory, you must use the `cd` command:

```
$cd [new directory path]
```

```
$cd ~
```

To create a directory called `test` in the root home folder, use the `mkdir` command:

```
$mkdir [new directory name]
```

To copy, move, and rename a directory, use the same command for the file commands. Sometimes you must add the `-r` (which stands for recursive) switch to involve the subdirectories as well:

```
$cp -r [source directory path] [destination directory path]
$mv -r [source directory path] [destination directory path]
$mv -r [original directory name] [new directory name]
```

To delete a folder, you must add the `-r` switch to the `rm` command to get the job done:

```
$rm -r [folder to delete path]
```

Mounting a Directory

Let's see a practical example of how to mount a directory inside Kali Linux. Let's suppose you inserted a USB key; then mounting a directory is necessary to access your USB drive contents. This is applicable if you disabled the auto-mount feature in your settings (which is on by default in the Kali 2020.1 release).

Searching and Filtering Text

One more thing to learn in the world of text files is the search mechanism. There are so many ways to search and filter out text, but the popular ones are as follows:

- `grep`
- `awk`
- `cut`

You've seen me using the `grep` command a lot. This filter command is structured in the following way:

```
$grep [options] [pattern] [file name]
```

Let's say you want to search for the word *password* in all the files starting from the root system (`/`).

```
root@kali:/# grep -irl "password" /
/boot/grub/i386-pc/zfscrypt.mod

/boot/grub/i386-pc/normal.mod
/boot/grub/i386-pc/legacycfg.mod
```

Here's what the options mean:

- `-i` : To ignore case and include all the uppercase/lowercase letters
- `-r` : To search recursively inside subfolders
- `-l` : To print the filenames where the filter matches

As another example, let's say you want to count the number of occurrences of the word *password* in the dictionary file `rockyou.txt` :

```
root@kali:/# cd /usr/share/wordlists/
root@kali:/usr/share/wordlists# grep -c "password" rockyou.txt
3959
```

The `awk` command is an advanced tool for filtering text files, and it uses the following pattern:

Figure 2.1 Bash Scripting

Let's divide the command so you can understand what's going on:

- `%s` : Means we're inserting a string (text) in this position
- `%d` : Means we're adding a decimal (number) in this position
- `\n` : Means that we want to go to a new line when the print is finished

Also, take note that we are using double quotes instead of single quotes. Double quotes will allow us to be more flexible with string manipulation than the single quotes. So, most of the time, we can use the double quotes for `printf` (we rarely need to use the single quotes).

To format a string using the `printf` command, you can use the following patterns:

- `%s` : String (texts)
- `%d` : Decimal (numbers)
- `%f` : Floating-point (including signed numbers)
- `%x` : Hexadecimal
- `\n` : New line
- `\r` : Carriage return
- `\t` : Horizontal tab

Variables

What is a variable, and why does every programming language use it anyway?

Consider a variable as a storage area where you can save things like strings and numbers. The goal is to reuse them over and over again in your program, and this concept applies to any programming language (not just Bash scripting).

To declare a variable, you give it a name and a value (the value is a string by default). The name of the variable can only contain an alphabetic character or underscore (other programming languages use a different naming convention). For example, if you want to store the IP address of the router in a variable, first you will create a file `var.sh` (Bash script files will end with `.sh`), and inside the file, you'll enter the following:

```
#!/bin/bash
#Simple program with a variable

ROUTERIP="10.0.0.1"

printf "The router IP address: $ROUTERIP\n"
```

Let's explain your first Bash script file:

- `#!/bin/bash` is called the *Bash shebang*; we need to include it at the top to tell Kali Linux which interpreter to use to parse the script file (we will use the same concept in Chapter 18, "Pentest Automation with Python," with the Python programming language). The `#` is used in the second line to indicate that it's a comment (a comment is a directive that the creator will leave inside the source code/script for later reference).
- The variable name is called `ROUTERIP` , and its value is 10.0.0.1.
- Finally, we're *printing* the value to the output screen using the `printf` function.

To execute it, make sure to give it the right permissions first (look at the following output to see what happens if you don't). Since we're inside the same directory (`/root`), we will use `./var.sh` to execute it:

```
root@kali:~# ./var.sh
bash: ./var.sh: Permission denied
root@kali:~# chmod +x var.sh
root@kali:~# ./var.sh
The router IP address: 10.0.0.1
```

Congratulations, you just built your first Bash script! Let's say we want this script to *run automatically* without specifying its path anywhere in the system. To do that, we must add it to the `$PATH` variable. In our case, we will add `/opt` to the `$PATH` variable so we can save our custom scripts in this directory.

First, open the `.bashrc` file using any text editor. Once the file is loaded, scroll to the bottom and add the line highlighted in Figure 2.2.

```
# Some more alias to avoid making mistakes:
# alias rm='rm -i'
# alias cp='cp -i'
# alias mv='mv -i'
export PATH=$PATH:/opt/
```

Figure 2.2 Export Config

and close all the terminal sessions. Reopen the terminal window and copy the script file to the `/opt` folder. From now on, we don't need to include its path; we just execute it by typing the script name `var.sh` (you don't need to re-execute the `chmod` again; the execution permission has been already set):

```
root@kali:~# cp var.sh /opt/
root@kali:~# cd /opt
root@kali:/opt# ls -la | grep "var.sh"

-rwxr-xr-x 1 root root          110 Sep 28 11:24 var.sh
root@kali:/opt# var.sh
The router IP address: 10.0.0.1
```

Commands Variable

Sometimes, you might want to execute commands and save their output to a variable. Most of the time, the goal behind this is to manipulate the contents of the command output. Here's a simple command that executes the `ls` command and filters out the filenames that contain the word *simple* using the `grep` command. (Don't worry, you will see more complex scenarios in the upcoming sections of this chapter. For the time being, practice and focus on the fundamentals.)

```
#!/bin/bash
LS_CMD=$(ls | grep 'simple')
printf "$LS_CMD\n"
```

Here are the script execution results:

```
root@kali:/opt# simplels.sh
simpleadd.sh
simplels.sh
```

Script Parameters

Sometimes, you will need to supply parameters to your Bash script. You will have to separate each parameter with a space, and then you can manipulate those params inside the Bash script. Let's create a simple calculator (`simpleadd.sh`) that adds two numbers:

```
#!/bin/bash
#Simple calculator that adds 2 numbers
```

```
#Store the first parameter in num1 variable
NUM1=$1
#Store the second parameter in num2 variable
NUM2=$2
#Store the addition results in the total variable
TOTAL=$(($NUM1 + $NUM2))

echo '########################'
printf "%s %d\n" "The total is =" $TOTAL
echo '########################'
```

You can see in the previous script that we accessed the first parameter using the `$1` syntax and the second parameter using `$2` (you can add as many parameters as you want).

Let's add two numbers together using our new script file (take note that I'm storing my scripts in the `opt` folder from now on):

```
root@kali:/opt# simpleadd.sh 5 2
########################
The total is = 7
########################
```

There is a limitation to the previous script; it can add only two numbers. What if you want to have the flexibility to add two to five numbers? In this case, we can use the default parameter functionality. In other words, by default, all the parameter values are set to zero, and we add them up once a real value is supplied from the script:

```
#!/bin/bash
#Simple calculator that adds until 5 numbers

#Store the first parameter in num1 variable
NUM1=${1:-0}
#Store the second parameter in num2 variable
NUM2=${2:-0}
#Store the third parameter in num3 variable
NUM3=${3:-0}
#Store the fourth parameter in num4 variable
NUM4=${4:-0}
#Store the fifth parameter in num5 variable
NUM5=${5:-0}
#Store the addition results in the total variable
TOTAL=$(($NUM1 + $NUM2 + $NUM3 + $NUM4 + $NUM5))
```

```
echo '########################'
printf "%s %d\n" "The total is =" $TOTAL
echo '########################'
```

To understand how it works, let's look at the `NUM1` variable as an example (the same concept applies to the five variables). We will tell it to read the first parameter `{1` from the terminal window, and if it's not supplied by the user, then set it to zero, as in `:-0}` .

Using the default variables, we're not limited to adding five numbers; from now on, we can add as many numbers as we want, but the maximum is five (in the following example, we will add three digits):

```
root@kali:~# simpleadd.sh 2 4 4
########################
The total is = 10
########################
```

TIP

If you want to know the number of parameters supplied in the script, then you can use the `$#` to get the total. Based on the preceding example, the `$#` will be equal to three since we're passing three arguments.

If you add the following line after the `printf` line:

```
printf "%s %d\n" "The total number of params =" $#
```

you should see the following in the terminal window:

```
root@kali:~# simpleadd.sh 2 4 4

########################

The total is = 10

The total number of params = 3

########################
```

User Input

Another way to interact with the supplied input from the shell script is to use the read function. Again, the best way to explain this is through examples. We will ask the user to enter their first name and last name after which we will print the full name on the screen:

```
 #!/bin/bash

read -p "Please enter your first name:" FIRSTNAME
read -p "Please enter your last name:" LASTNAME

printf "Your full name is: $FIRSTNAME $LASTNAME\n"
```

To execute it, we just enter the script name (we don't need to supply any parameters like we did before). Once we enter the script's name, we will be prompted with the messages defined in the previous script:

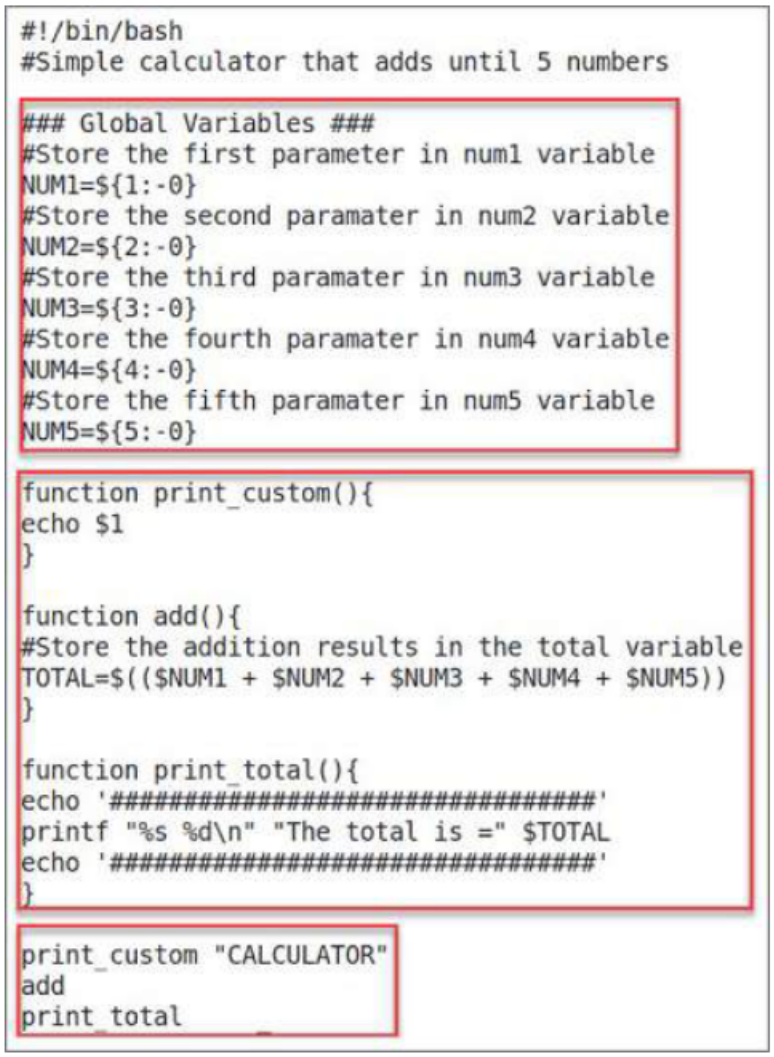
```
root@kali:~# nameprint.sh
Please enter your first name:Gus
Please enter your last name:Khawaja
Your full name is: Gus Khawaja
```

Functions

Functions are a way to organize your Bash script into logical sections instead of having an unorganized structure (programmers call it *spaghetti code*). Let's take the earlier calculator program and reorganize it (refactor it) to make it look better.

This Bash script (in Figure 2.3) is divided into three sections:

- In the first section, we create all the global variables. Global variables are accessible inside any function you create. For example, we are able to use all the `NUM` variables declared in the example inside the `add` function.
- Next, we build the functions by dividing our applications into logical sections. The `print_custom()` function will just print any text that we give it. We're using the `$1` to access the parameter value passed to this function (which is the string `CALCULATOR`).
- Finally, we call each function sequentially (each one by its name). Print the header, add the numbers, and, finally, print the results.



```
#!/bin/bash
#Simple calculator that adds until 5 numbers

### Global Variables ###
#Store the first parameter in num1 variable
NUM1=${1:-0}
#Store the second paramater in num2 variable
NUM2=${2:-0}
#Store the third paramater in num3 variable
NUM3=${3:-0}
#Store the fourth paramater in num4 variable
NUM4=${4:-0}
#Store the fifth paramater in num5 variable
NUM5=${5:-0}

function print_custom(){
echo $1
}

function add(){
#Store the addition results in the total variable
TOTAL=$(($NUM1 + $NUM2 + $NUM3 + $NUM4 + $NUM5))
}

function print_total(){
echo '################################'
printf "%s %d\n" "The total is =" $TOTAL
echo '################################'
}

print_custom "CALCULATOR"
add
print_total
```

Figure 2.3 Script Sections

Conditions and Loops

Now that you know the basics of Bash scripting, we can introduce more advanced techniques. When you develop programs in most programming languages (e.g., PHP, Python, C, C++, C#, etc.), including Bash scripting, you will encounter conditions (`if` statements) and loops, as shown in Figure 2.4.

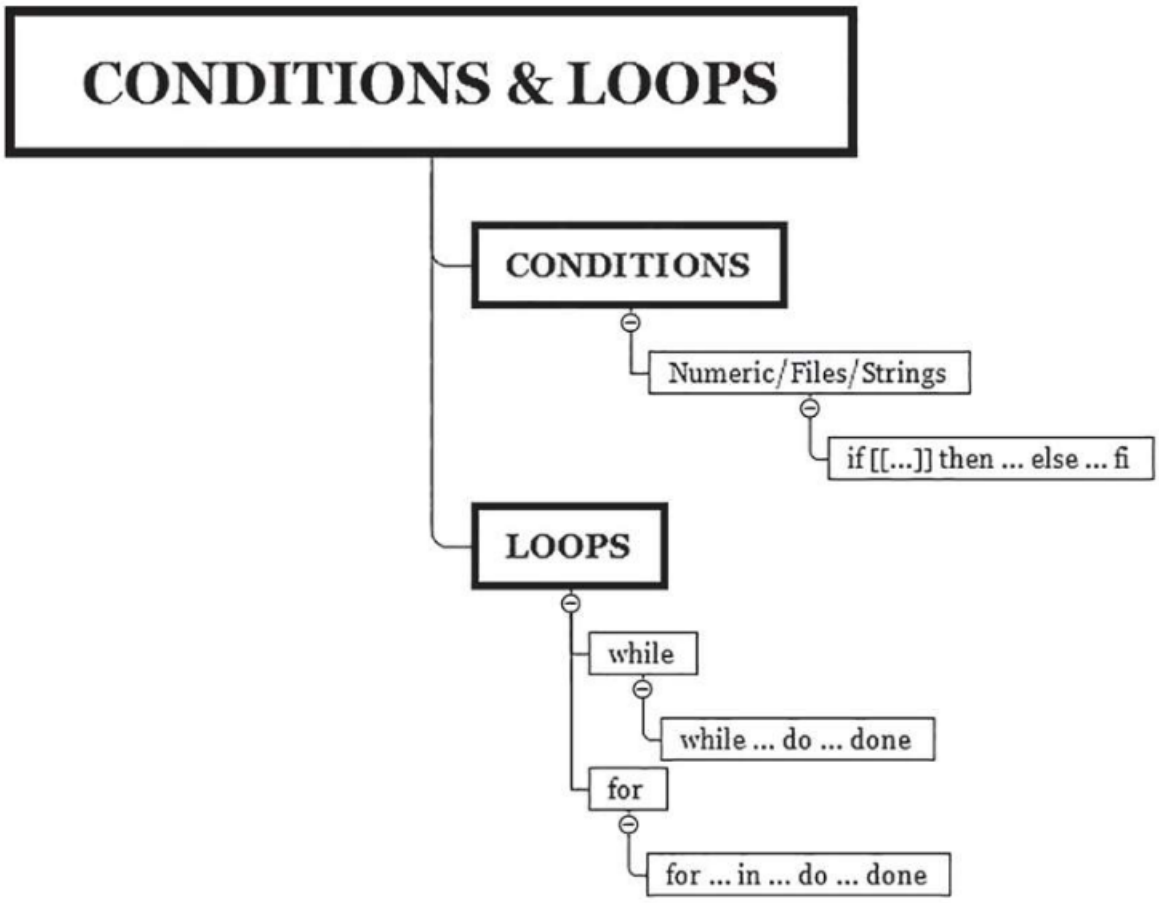


Figure 2.4 Conditions and Loops

Conditions

An `if` statement takes the following pattern:

```
if [[ comparison ]]
then
True, do something
else
False, Do something else
fi
```

If you've been paying attention, you know that the best way to explain this pattern is through examples. Let's develop a program that pings a host using Nmap, and we'll display the state of the machine depending on the condition (the host is up or down):

```
#!/bin/bash
#Ping a host using Nmap
```

```
### Global Variables ###
#Store IP address
IP_ADDRESS=$1

function ping_host(){
ping_cmd=$(nmap -sn $IP_ADDRESS | grep 'Host is up' | cut -d '(' -f
1)
}

function print_status(){
if [[ -z $ping_cmd ]]
then
echo 'Host is down'
else
echo 'Host is up'
fi
}

ping_host
print_status
```

The `nmap` command either returns an empty string text if the host is down or returns the value “Host is up” if it's responding. (Try to execute the full `nmap` command in your terminal window to visualize the difference. If so, replace `$IP_ADDRESS` with a real IP address.) In the `if` condition, the `-z` option will check if the string is empty; if yes, then we print “Host is down” or else we print “Host is up:”

```
root@kali:~# simpleping.sh 10.0.0.11
Host is down
root@kali:~# simpleping.sh 10.0.0.1
Host is up
```

What about other condition statements? In fact, you can compare numbers, strings, or files, as shown in Tables 2.1, 2.2, and 2.3.

Table 2.1 Numerical Conditions

Equal	`[[ x -eq y ]]`
Not equal	`[[ x -ne y ]]`
Less than	`[[ x -lt y ]]`
Greater than	`[[ x -gt y ]]`

Table 2.2 String Conditions

Equal	`[[ str1 == str2 ]]`
Not equal	`[[ str1 != str2 ]]`
Empty string	`[[ -z str ]]`
Not empty string	`[[ -n str ]]`

Table 2.3 File/Directory Conditions

File exists?	`[[ -a filename ]]`
Directory exists?	`[[ -d directoryname ]]`
Readable file?	`[[ -r filename ]]`
Writable file?	`[[ -w filename ]]`
Executable file?	`[[ -x filename ]]`
File not empty?	`[[ -s filename ]]`

Loops

You can write loops in two different ways: using a `while` loop or using a `for` loop. Most of the programming languages use the same pattern for loops. So, if you understand how loops work in Bash, the same concept will apply for Python, for example.

Let's start with a `while` loop that takes the following structure:

```
while [[ condition ]]
do
do something
done
```

The best way to explain a loop is through a counter from 1 to 10. We'll develop a program that displays a progress bar:

```
#!/bin/bash
#Progress bar with a while loop

#Counter
COUNTER=1
#Bar
BAR='##########'
```

```
while [[ $COUNTER -lt 11 ]]
do
#Print the bar progress starting from the zero index
echo -ne "\r${BAR:0:COUNTER}"
#Sleep for 1 second
sleep 1
#Increment counter
COUNTER=$(( $COUNTER +1 ))
done
```

Note that the condition (`[[ $COUNTER -lt 11]]`) in the `while` loop follows the same rules as the `if` condition. Since we want the counter to stop at 10, we will use the following mathematical formula: `counter<11` . Each time the counter is incremented, it will display the progress. To make this program more interesting, let it sleep for one second before going into the next number.

On the other hand, the `for` loop will take the following pattern:

```
for … in [List of items]
do
something
done
```

We will take the same example as before but use it with a `for` loop. You will realize that the `for` loop is more flexible to implement than the `while` loop. (Honestly, I rarely use the `while` loop.) Also, you won't need to increment your index counter; it's done automatically for you:

```
#!/bin/bash
#Progress bar with a For Loop

#Bar
BAR='##########'

for COUNTER in {1..10}
do
#Print the bar progress starting from the zero index
echo -ne "\r${BAR:0:$COUNTER}"
#Sleep for 1 second
sleep 1
done
```

File Iteration

Here's what you should do to simply read a text file in Bash using the `for` loop:

Gathering data using WHOIS

What if you can access a database that contains the records of registered domains on the internet? Many domain registrars allow the general public to view publicly accessible information about domains. This information can be found on various **WHOIS** databases on the internet.

The following is a brief list of some information types that are usually stored for public records:

- Registrant contact information
- Administrative contact information
- Technical contact information
- Name servers