Android User Enabled Security: Passwords and Gesture

Information in This Chapter

- Security on Androids
- Simple security values
- The password lock
- Hashcat
- The pattern lock (gesture)
- Using a rainbow table
- SHA-1 exercise

Introduction—Security on Androids

Since their inception, Android devices (the first being the HTC G1) have provided the user the ability to employ simple security measures. As the operating system versions changed, different types of security also evolved. Today, the consumer can rely on embedded security features from a specific OS or exclusive settings based on the make and model. There is also an ability to download and install specific applications for the same purpose. Some of the more popular and embedded by OS and/or make and model, include but are not limited to:

- PIN
- Password
- Pattern (gesture)
- Fingerprint
- Facial recognition
- Knock lock

It would be unnecessary to dedicate an entire chapter showing each of these security types. Instead, this chapter will primarily address the password and gesture lock. This is continuing to be one of the more popular user enabled security measures employed on Androids. Some of you may be reading this and saying, "My _____ (fill in the blank) unit/machine/utility can get past that security. What do I need to know more about this process?"

Yes, in the forensic utility market, many makes and models *are* supported for bypassing the gesture lock. There are some examiners, however, who may find the need to testify on the process or, in some cases, manually obtain the lock value(s) to enter the phone for visual validation, manual documentation, or to enable/disable a setting to perform an additional examination. The content in this chapter will allow the examiner to have more insight into at least one of the more popular security measures that is stored on an Android.

Simple Security Values

Before the advent of smartphones, the operating system on some phones would simply store the actual user value within the operating system. This means that if the user placed a handset lock code of 1325 on a simple bar or flip phone, that value would be stored in the basic encoding for the phone, usually in ASCII or Unicode. It could easily be located by employing encoding search techniques using popular tools such as Cellebrite's Physical Analyzer. Also, many of the devices will contain the user PIN and user security code just a few offsets from one another within the file system. On some phones, the security code is used to bypass the PIN if it is unknown. In a past criminal investigation several years ago, a homicide victim decided to change his 4-digit PIN and 6-digit (default) security codes. The phone was a Motorola iDEN. At the time of this incident, automatic tools were limited, and large vendors such as Cellebrite did support the phone for physical extraction, but the parsing on SMS and other user areas was limited. Logical settings, however, would parse the required fields for the case. A test phone of the same model was used to place a different (known) value for the PIN and 6-digit security code. Using the *Find—Code* tab in Physical Analyzer, the values on the test phone were easily located. They were an offset apart and had unique characters before and after these security codes. A RegEX expression was then used to locate the values on the victim's phone, unlock the GUI, and allow a logical parsing of several SMS messages related to his death. The point of explaining this case is to convey that these values were on the phone, in their original state. This was typically how phones stored the security imposed by the user. There are many phones still sold today that work this way. These generally fall into the pay-as-you-go, simple "burner-phone" categories. They lack sophisticated operating systems and have limited features. Many are supported by several commercial forensic tools, while others may lock down their USB data port.

Fig. 19.1 is a file system from an LG VX8100 that was acquired from Bitpim. Within the *nvm_security* folder we can see the default 6-digit security value of *000000*. The user enabled security PIN is below this, toward the end of offset 90 as *1397* (bottom image in Fig. 19.1).

[] r	nvm <u>.</u>	sec	urity	/						1	Reg	ular	File		6	/25/2008 3:12:
00	00	00	00	00	00	00	00	00-00	00	00	00	00	00	00	00	
10	00	00	00	00	00	00	00	00-00	00	00	00	00	00	00	00	
20	00	00	00	00	00	00	00	00-00	00	00	00	00	00	00	00	
30	00	00	00	00	00	00	00	00-00	00	00	00	00	00	00	00	
40	00	00	00	00	00	00	00	00-00	00	00	00	00	00	00	00	
50	00	00	00	00	00	00	00	00-00	00	00	00	00	00	00	00	
60	00	00	00	00	00	00	00	00-00	00	01	FF	FF	01	FF	FF	·····ÿÿ ·ÿÿ
70	01	FF	FF	01	30	30	30	30-30	30	00	00	00	00	00	00	-ÿÿ - <mark>000000</mark>
80	00	00	00	00	00	00	00	00-00	00	00	00	00	01	00	01	
90	00	01	FF	FF	FF	FF	01	FF-FF	FF	FF	01	00	01	31	33	··ÿÿÿÿ·ÿÿÿÿÿ···13
a 0	39	37	01	01	01	00	00	01-00	00	00	00	00	00	00	00	97 • • • • • • • • • • • • • • • • • • •
b0	00	00	00	00	00	00	00	00-00	00	00	00	00	00	00	01	
c 0	00	00	00	00	00	00	00	00-00	00	00	00	01	00	00	00	•••••
d0	00	00	00	00	00											

Six-digit security (Non-smart phone OS)

40	00	00	00	00	00	00	00	00-00	00	00	00	00	00	00	00	
50	00	00	00	00	00	00	00	00-00	00	00	00	00	00	00	00	
60	00	00	00	00	00	00	00	00-00	00	01	FF	FF	01	FF	FF	ÿÿ.ÿÿ
70	01	FF	FF	01	30	30	30	30-30	30	00	00	00	00	00	00	·ÿÿ ·000000 · · · · ·
80	00	00	00	00	00	00	00	00-00	00	00	00	00	01	00	01	
90	00	01	FF	FF	FF	FF	01	FF-FF	FF	FF	01	00	01	31	33	·· ÿÿÿÿ ·ÿÿÿÿ··· <mark>13</mark>
a0	39	37	01	01	01	00	00	01-00	00	00	00	00	00	00	00	97
b0	00	00	00	00	00	00	00	00-00	00	00	00	00	00	00	01	
c0	00	00	00	00	00	00	00	00-00	00	00	00	01	00	00	00	
d0	00	00	00	00	00											
	-															-

Four-digit user enabled security (Non-smart phone OS)

Figure 19.1

Example of default security and user set PIN values of an LG VX8100 mobile phone.

Smartphones

In most cases when the user employs a specific type of security on a smartphone, the actual value is converted out of this simple format shown in Fig. 19.1. Depending on which type of security the user chooses, it runs a check against a specific hash value of that security when the user wants to enter the GUI. It is important that examiners understand that when it comes to the gesture and password locks, we are not addressing the entire file

system encryption, but simply the security (value) encryption that is performed each time a user enters the phone. This is not to say that there is no hardware-based encryption that may be going on with newer chip sets. Many readers are probably aware of issues with newer iPhones. This also applies to newer devices using the UFS chip. This will become even more problematic, and new techniques will hopefully address exploits for these issues.

This chapter will discuss the security aspect of both the *password lock* and the *pattern lock* employed on Androids. It will conclude with focusing on manually creating a hash value based on a binary file created from the gesture pattern.

The Password Lock

When a user sets up a password lock, he/she can employ his/her own choice of data that contain special characters, numbers, letters, or any combination of the same. If the user sets up a password lock, it is given a hash that uses a combination of SHA-1 and MD5. The value is also salted. Salt or salting is the process of adding additional security, as it randomizes the hash. This can prevent dictionaries and rainbow tables from breaking the hash values that are always the same, such as what readers will see when we discuss the gesture value. The Android password can be complex, with 4–16 characters in length. There are 94 possible characters per space, with users having the choice of lower/uppercase letters, digits, and punctuation. The good news (if it can be called that) is that there are no spaces allowed. If we manually locate the value within the file password.key (Data/System/) location, the password hash has a 72-byte hexadecimal value. From left to right, it is comprised of the first 40 bytes showing the SHA-1 hash. This is immediately followed by the remaining 32 bytes of the MD5 hash. Because the value is salted, the salt value must be recovered first. Salt is represented in hexadecimal as a random 64-bit integer. Salt is stored in *settings.db* (SQLite database file), which is named *lockscreen.password_salt*. This will be within the (rebuilt) file system shown here:

data/data/com.android.providers.settings/databases/settings.db

If examiners have pulled a binary that has not been rebuilt and decoded with tools such as Physical Analyzer, it is a little bit more time-consuming to locate the password.key and the salt value. If examiners conduct an ASCII search for "lockscreen.password_salt," they will obtain at least one hit that places them in general the area they need to look for the entire salt value. The salt is a string of the hexadecimal representation of a random 64-bit integer. The value along with the MD5 or SHA-1 from the password key location can then be used to brute force attack the values with Hashcat. Here are some steps to help examiners locate the salt when they have a physical file from a JTAG, ISP, or chip-off examination.

Hashcat

Hashcat is a free, open-source tool, which supports several different algorithms, and can be installed in multiple operating systems [1]. Using their latest list from the version available at the time this was written (3.30), the following algorithms can be attacked:

MD4, MD5, Half MD5 (left, mid, right), SHA1, SHA-224, SHA-256, SHA-384, SHA-512, SHA-3 (Keccak), SipHash, RipeMD160, Whirlpool, DES (PT=\$salt, key=\$pass), 3DES (PT=\$salt, key=\$pass), GOST R 34.11-9, GOST R 34.11-2012 (Streebog) 256-bit, GOST R 34.11-2012 (Streebog) 512-bit, Double MD5, Double SHA1, md5(\$pass.\$salt), md5(\$salt.\$pass), md5(unicode(\$pass).\$salt), md5(\$salt. unicode(\$pass)), md5(sha1(\$pass)), md5(\$salt.md5(\$pass)), md5(\$salt.\$pass.\$salt), md5(strtoupper(md5(\$pass))), sha1(\$pass.\$salt), sha1(\$salt.\$pass), sha1(unicode(\$pass).\$salt), sha1(\$salt.unicode(\$pass)), sha1(md5(\$pass)), sha1(\$salt.\$pass.\$salt), sha1(CX), sha256(\$pass.\$salt), sha256(\$salt.\$pass), sha256(unicode(\$pass).\$salt), sha256(\$salt.unicode(\$pass)), sha512(\$pass.\$salt), sha512(\$salt.\$pass), sha512(unicode(\$pass).\$salt), sha512(\$salt.unicode(\$pass)), HMAC-MD5 (key=\$pass), HMAC-MD5 (key=\$salt), HMAC-SHA1 (key=\$pass), HMAC-SHA1 (key=\$salt), HMAC-SHA256 (key=\$pass), HMAC-SHA256 (key=\$salt), HMAC-SHA512 (key=\$pass), HMAC-SHA512 (key=\$salt), PBKDF2-HMAC-MD5, PBKDF2-HMAC-SHA1, PBKDF2-HMAC-SHA256, PBKDF2-HMAC-SHA512, MyBB, phpBB3, SMF, vBulletin, IPB, Woltlab Burning Board, osCommerce, xt:Commerce, PrestaShop, Mediawiki B type, Wordpress, Drupal, Joomla, PHPS, Django (SHA-1), Django (PBKDF2-SHA256), EPiServer, ColdFusion 10+, Apache MD5-APR, MySQL, PostgreSQL, MSSQL, Oracle H: Type (Oracle 7+), Oracle S: Type (Oracle 11+), Oracle T: Type (Oracle 12+), Sybase, hMailServer, DNSSEC (NSEC3), IKE-PSK, IPMI2 RAKP, iSCSI CHAP, Cram MD5, MySQL Challenge-Response Authentication (SHA1), PostgreSQL Challenge-Response Authentication (MD5), SIP Digest Authentication (MD5), WPA, WPA2, NetNTLMv1, NetNTLMv1+ESS, NetNTLMv2, Kerberos 5 AS-REQ Pre-Auth etype 23Kerberos 5 TGS-REP etype 23, Netscape LDAP SHA/SSHA, LM, NTLM, Domain Cached Credentials (DCC), MS Cache, Domain Cached Credentials 2 (DCC2), MS Cache 2, MS-AzureSync PBKDF2-HMAC-SHA256, descrypt, bsdicrypt, md5crypt, sha256crypt, sha512crypt, bcrypt, oSX v10.4, OSX v10.5, OSX v10.6, OSX v10.7, OSX v10.8, OSX v10.9, OSX v10.10, AIX {smd5}, AIX {ssha1}, AIX {ssha256}, AIX {ssha512}, Cisco-ASA, Cisco-PIX, Cisco-IOS, Cisco \$8\$, Cisco \$9\$, Juniper IVE, Juniper Netscreen/SSG (ScreenOS), Android PIN, Windows 8+phone PIN/Password, GRUB 2, CRC32, RACF, Radmin2, Redmine, OpenCart, Citrix Netscaler, SAP CODVN B (BCODE), SAP CODVN F/G (PASSCODE), SAP CODVN H (PWDSALTEDHASH) iSSHA-1, PeopleSoft, PeopleSoft PS_TOKEN, Skype, WinZip, 7-Zip, RAR3-hp, RAR5, AxCrypt, AxCrypt in

memory SHA1, PDF 1.1–1.3 (Acrobat 2–4), PDF 1.4–1.6 (Acrobat 5–8), PDF 1.7 Level 3 (Acrobat 9), PDF 1.7 Level 8 (Acrobat 10–11), MS Office <= 2003 MD5, MS Office <= 2003 SHA1, MS Office 2007, MS Office 2010, MS Office 2013, Lotus Notes/Domino 5, Lotus Notes/Domino 6, Lotus Notes/Domino 8, Bitcoin/Litecoin wallet.dat, Blockchain, My Wallet, 1Password, agilekeychain, 1Password, cloudkeychain, Lastpass, Password Safe v2, Password Safe v3, Keepass 1 (AES/Twofish) and Keepass 2 (AES), Plaintext, eCryptfs, Android FDE <= 4.3, Android FDE (Samsung DEK), TrueCrypt, VeraCrypt [1].

The use of Hashcat could easily take up a couple of different chapters. We will not spend too much time covering the exact steps of this program. In previous versions, the user needed to specify if they needed oclHashcat, for AMD graphic cards, or CudaHashcat, if they used Nvidia. They have combined the use into one program, and some of the examples used in this chapter used version 3.30.

When we attempt to break the password on an Android, Hashcat has the ability to break the entire string of 72 bytes, along with the salt, but the time evolved tremendously increases. If we focus on separating these hashes, and then locating the salt, it makes the attack much easier to manage. Of course, the longer the password, the more time needed. Most individuals do not use all 16 characters and generally keep their password relatively short. The food for thought here is if you have a robust forensic machine, why not try it? Hashcat usually runs an estimated time on how long the process will take. From there you can decide if you want to continue or keep it running and hand it over to your replacement after you retire with 30 years of service.

If we are dealing with a raw physical pull that has not been decoded, we can use the search technique previously described (*lockscreen.password_salt*) to locate the general area to find the salt. As we look at the search results, we need to examine the byte value directly in front of this search hit. It will be in a range between $0\times0F$ and 0×35 . That byte provides the examiner the length of the salt. Directly in front of these series of bytes, there will be a byte displaying the value of $0\times3D$. This byte represents a string length of 24, which is the length of our value we are looking for. Fig. 19.2 depicts an actual salt value that represents these values we just explained and the ASCII search hit that was used. For readers who are viewing this book electronically, they are color coded and labeled to illustrate this further. Once we locate the salt integer, it needs to be converted to hexadecimal. In older versions of Hashcat, the uppercase letters in the converted value must be converted to lowercase to conduct brute force attempts. The newer version of Hashcat (3.30) no longer requires this step. Fig. 19.3 depicts the salt value from the *settings.db* area, which is much easier to locate. Again, depending on which type of extraction examiners have performed, the method needed for locating the salt will be determined. Obviously, the database is much easier and preferred, if this is possible in your casework.

															AS	CII	sea	rch	: " lo	ck	scre	en.	.pas	sword_salt"
		5	7	65	6C	6C	43	68	65	63	6B	65	7	53	65	72	76	69	63	65	0E	99	00	dSpellCheckerService
7	- 7	ς.	$\overline{\mathbb{A}}$	13	65	09		00			25		A	ΖA	81	51	04	00	37	OF	6C	6F	63	.G.se%.bQ7.loc
6	<u>v</u>	5V/	70	61	74	74	65	72	6E	5F	61	75	74	6F	6C	6F	63	6B	30	30	81	50	04	k_pattern_autolock00.P.
0	0	ЗD	35	6C	6F	63	6B	73	63	72	65	65	6E	2E	70	61	73	73	77	6F	72	64	5F	.5lockscreen.password
7	3	61	6C	74	2D	35	32	37	38	30	30	39	39	32	36	34	35	36	32	38	37	38	35	<mark>salt</mark> -527800992645628785
3	0	1D	81	4F	04				53		53	54	45	4D			48						41	
5	3	53	57	4F	52	44	31	33	32	35	2в	81	29	04	00	47	21	73	65	6C	65	63	74	SSWORD1325)G!select

Salt value (integer) must be converted to a hexadecimal value. The upper letters are

lower cased for HashCat to brute force (b6c0bdef9c965d96)

tw Edit I	Help							
		-52	2786	0099	9264	156	2878	350
1011 01 63 1001 11 31			0000	1011 47 0101 15	110			1111 32 0110 0
🖯 Hex		Med	A	MC	MR	MS	M+	M
Dec Oct	(В	-	α	c		4
0 Bin	RoL	RoR	С	7	8	9	1	%
Qword	Or	Xor	D	4	5	6	•	1/8
Owerd Word	Lsh	Rsh	E	1	2	3	-	
Dyte	Not	And	E		0		-	=

			B	5008	BDER	909	9650	09
63			0000 0110	1011 47 0101 15	110			111 3 011
Hex		Mod	A	MC	MR	MS	M+	•
Oct	(В	-	CE	¢	:	
() Bin	RoL	RoR	С	7	8	9	1	
Qword	Or	Xor	D	4	5	6		1
O Dword	Lsh	Rsh	E	1	2	3		1
O Dyte	Not	And	E		0		+	



Breakdown of locating the salt value in Hex View and converting the integer value to hexadecimal.

settings.dl	0				Calculator	Melo					E		
Database view		View	File Info										
Database view	Hex	View	File Info				-53	2786	0099	264	1562	878	350
													_
android_metadata (1) bluetooth_devices (0)	Ø	_id •	name	value	1011 01 63 1001 11 31			0000	1011 47 0101 15	110			1111 32 0110 0
bookmarks (8)	2	44	voice_recognition_service	com.vlingo.client.samsung/com.vli	Ottex		Med	A	MC	MR	MS	M+	M-
secure (62) sqlite_sequence (2)		45	throttle_reset_day	9	@ Dec								
system (157)		46	data_national_roaming_mode	0	Ood	(В	-	ct	c 9 6		
.,		48	device_provisioned	1	Otin	RoL	RoR	С	7	8		1	%
		49	last_setup_shown	eclair_1	@ Qword	Or	Xee	D	4	5		-	1/x
	2	52	preferred_tty_mode	0	Opword	-			-				-
		53	tty_mode_enabled	0	OWord	Lsh	Rsh	E	1	2	3	-	-
		59	dropbox:data_app_wtf	disabled	🔿 Byte	Not	And	F	0			+	-
		60	ssl_session_cache	file									
		61	facelock_detection_threshold	0.0									
		62	dropbox:data_app_crash	disabled									
		63	send_action_app_error	1		_	_	_	_	_		-	_
		64	dropbox:data_app_anr	disabled	Calculator							0 0	- 0 🔜
		65	web_autofill_query_url	http://android.clients.google.com/	View Edit	Help							
		66	backup_enabled	1									
		67	backup_provisioned	1				B	5CØE	BDEF	-9C	9650	096
		166	input_methods_subtype_history	com.google.android.inputmethod.	1011 01	10 1	100	00.00	1011	110	1 11	10	1111
		169	selected_input_method_subtype	-921088104	63 1001 11			0110	47 0101	110			32 0110
		170	default_input_method	com.google.android.inputmethod.	31			0110	15	110		101	0
	2	183	allowed_geolocation_origins		@ Hex		Med	A	MC	MR	MS	M-	M.
		187	bluetooth_on	0	ODec								
		199	location_providers_allowed		Ood	(В	-	α	c	1	4
		206	wifi_on	0	© 8in	RoL	RoR	С	7	8	9	-1-	%
		207	SYSTEM_PHONE_PASSWORD	1325	@ Qword	Or	Xor	D	4	5	6	-	1/x
	V	208	lockscreen.password_salt	-5278009926456287850	Qword Dword				-		-	<u> </u>	-/1
				0	OWord	Lsh	Rsh	E	1	2	3	-	-
		211	lockscreen.password_type_alternate	0	O Dyte	Not	And	F	0			+	-
		212	lockscreen.password_type	131072									

Using the *settings.db* to obtain the salt value. Use the same conversation of the integer to hexadecimal. Use lower case letters for HashCat brute force (b6c0bdef9c965d96)

Figure 19.3

Breakdown of locating the salt value from the com.android.providers.settings/databases/settings. db location and converting the integer value. Hashcat can attempt brute force with the SHA-1 or MD5 followed by the salt value. Use one or the other, but to save time, not the entire combined 72-byte (SHA-1/MD5) password combination. From left to right, the first 40 bytes are the SHA-1, followed by the remaining bytes for MD5. Since the MD5 is shorter (32 bytes), it is much easier to use that value from the *password.key* location to attempt the attack.

The first four screenshots (figures) contained on the companion site deal directly with the help list from Hashcat. The first figure, Fig. 19.1 (companion site), is obtained from Hashcat 3.30. This is directly from the help command (C:\>hashcat64.exe -h), which displays the various options and commands that can be used. This first screenshot is just some of the choices, there are more. Fig. 19.1 (companion site) shows the first choice, which is the "Options." Fig. 19.2 (companion site) is a few of the "Hash modes." Again, not the entire list of all the choices. Fig. 19.3 (companion site) depicts the "Attack Modes." The last figure from Hashcat 3.30 (companion site, Fig. 19.4) shows some "Basic Examples" and links to where people can locate additional help.

Obviously, the tool runs in the command line (*Run As Administrator*) and, depending on the video card used, will represent which additional line codes a user would use. If we install Hashcat in the root of C: drive in Windows, we could use a command line on a 64-bit Windows machine that is running a supported GPU as the following:

C:\.Hashcat64.exe-a3-m110 AE36A990BF8300123E43EBF01E39EE41335F1406: b6c0bdef9c965d96

Using the embedded Hashcat general help list would show that $-a \ 3$ is a brute force attack option, and qualifier noted as -m is our option for hash type, and the 110 is used for SHA-1 (\$pass.\$salt). If we had used the MD5 value, the command would be the same except we would change the 110 to 10 for the MD5 (\$pass.\$salt).

This combination of salting the hash value made from a SHA-1 and MD5 is not always followed in every Android. Samsung, for example, changed some of the rules. Samsung III and higher no longer utilize the SHA-1 and MD5 combination and instead take the hash and salt and perform 1024 SHA-1 interactions.

Using John Lehr's "Android Password Possibilities" chart [2], let us put this into perspective with a length and number of password possibilities comparison:

Length (PW)	Number of (PW) possibilities
4	78,074,896
5	3,339,040,224
6	689,869,781,056
7	64,847,759,419,264
8	6,095,689,385,410,816

Android User Enabled Security: Passwords and Gesture 291

Length (PW)	Number of (PW) possibilities
9	572,994,802,228,616,704
10	53,861,511,409,489,970,176
11	5,062,982,072,492,057,196,544
12	475,920,314,814,253,376,475,136
13	44,736,509,592,539,817,388,662,784
14	4,205,231,901,698,742,834,534,301,696
15	395,291,798,759,681,826,446,224,359,424
16	37,157,429,083,410,091,685,945,089,785,856

As we can see, tools such as Hashcat are needed to speed up the process of brute force cracking passwords. Readers can locate additional information regarding Hashcat through Internet searches. Another interesting way some examiners are tackling the problem, is by daisy chaining several high-end graphic cards and create a machine dedicated to cracking passwords.

Q. Why decode the password lock?

In most forensic scenarios, the examiner has performed a physical pull, and for various case-related reasons, decoding of the password lock is needed. It may be a case where a file system or logical pull is necessary, and the examiner must get into the GUI of the device to enable the USB debugging. There may also be cases where the examiner needs to view application data under an exigent situation or validate data. Whatever the reason, there may be times when the physical examination is not enough for the case requirements.

The actual pull itself when there is a password lock enabled can be accomplished with rooted Androids, debugging being enabled already, or in some cases using JTAG or ISP pulls. There are some newer Androids that will not support rooting and/or JTAP/ISP. Because of the complexity of some passwords, python scripts could take in some cases, over 8500 millennia to complete. However, forensic machines that employ higher-end graphic card or multiple cards could crack the same password in hours. There are specific limitations and there may be situations where the passcode cannot be brute forced in the examiner's lifetime or at least without more hardware costs being employed on the host machine.

The Pattern Lock (Gesture)

The GUI of the Android shows the pattern lock as a series of round dots as shown in the simple illustration in Fig. 19.4. There are totally nine—three in three rows. If we number each of these and begin counting from left to right and continue counting left to right, each will have its own number. Like the password lock, the security does not store the values of each of these as a number, but SHA-1 hashes the value, and stores the SHA-1. Users who create a pattern are not allowed to move over one point several times, and there are only 895,924 variants to the pattern on an Android. The hashed SHA-1 value given to the bytes

is placed in the file system in the *gesture.key* file, again with the same path as before: / data/system. Computers begin counting at value 0. The first gesture is not a value of 1 converted to bytes then SHA-1, but rather a value of 0. This means that the values start at zero and end with eight, not nine. The hexadecimal equivalent of each single point is 0×00 , through 0×08 .

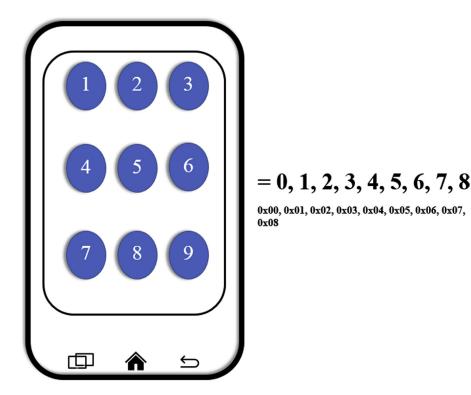




Illustration to show the numbering of the 9 pattern (*dots*) on the screen of an Android and how they are counted in hexadecimal.

If we created the minimum length of a gesture pattern of four areas and begin with the numbered values used 1,2,3,4 (left side image in Fig. 19.5), this would utilize corresponding bytes 00 01 02 03, which would in turn perform a SHA-1 hash. The resulting hash algorithm would be stored on the Android in the *gesture.key* location as:

A02A05B025B928C039CF1AE7E8EE04E7C190C0DB

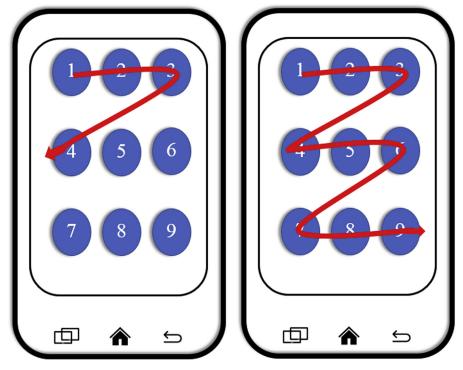


Figure 19.5 Two different examples of gesture security patterns (1,2,3,4 and 1,2,3,4,5,6,7,8,9).

If we create the gesture combination of 9,8,7,6,5,4,3,2,1, it would show as what is depicted in the right side image in Fig. 19.5, and convert the bytes to the SHA-1 value, which is also stored in the *gesture.key* location as:

853822DCEE4C6B59D4A9F0C4CDAF97989E29C83A

Extraction Summary

There are commercially available forensic programs that can decode the gesture pattern and display it in the summary of the report. For example, if we decode the gesture pattern on an LG LS-670, it may display within the summary as unlock pattern 1->2->3->6->9->8->5->4->7.

SHA-1 Exercise

To help better understand the values that are stored in the gesture.key location, there is a simple exercise that replicates the stored gesture that is hash as a SHA-1. Several of the tools

used in this example are free. The example also uses Physical Analyzer to view the file after we create it. Viewing the file that was created is not necessary but it does help validate that no mistakes were made with the bytes that were created. Also, other free viewers such as FTK Imager will work. To start with, download a free hex creation tool. The one that is shown in this example is called Hex Editor XVI32 and is created by Christian Mas. XVI32 is a freeware hex editor running under Windows 9x/NT/2000/XP/Vista/7. The name XVI32 is derived from *XVI*, the roman notation for the number 16 [3].

Once Hex Editor XVI32 is downloaded and installed, go to *Edit—Insert String*. This will bring up the Insert window. Leave the **Text string** and "**as Unicode Latin (UTF-16LE)**" settings blank. Select the *Hex string* setting. Leave all the other settings in their default settings as seen when the program first opens. In the blank box below the **Hex string** setting, type a simple gesture pattern of 00 01 02 03. The field will automatically space the bytes accordingly. This represents a user creating a pattern lock from left to right on the GUI of the phone of 1,2,3,4 as shown in our previous example (image) contained in the left side of Fig. 19.5. Once this has been completed, select **OK**. The file must be saved. Select *File—Save as*, and name the file. In the example provided, the file was named 01_ through_04. In the "Save as type" make certain (*.*) is selected. On the companion site, Fig. 19.5 depicts the first two steps when creating the file using the Hex Editor XVI32 program. Fig. 19.6 of the companion site also shows the naming configuration the file needs to be saved in.

This file can be opened with a hex viewer or FTK Imager. The next step is to create a SHA-1 hash value. Again, there are several free programs available, and FTK Imager can create both an MD5 and SHA-1, exporting the values to a .csv file. In our example, we use DigitalVolcano Hash Tool 1.1. "A freeware utility to calculate the hash of multiple files. This is a 128-bit number usually expressed as a 32-character hexadecimal number. It can be said to be the 'signature' of a file or string and is used in many applications, including checking the integrity of downloaded files. This compact application helps you quickly and easily list the hashes of your files" [4]. This free utility allows the user to hash desired files using MD5, SHA-1, SHA-256, SHA-384, SHA-512, and CRC-32 algorithms.

Once the program is downloaded and installed, we select SHA-1, and then select our saved *. * file, and in our example of 1,2,3,4 (bytes 00 01 02 03), the SHA-1 value created is:

a02a05b025b928c039cf1ae7e8ee04e7c190c0db

If we take a dictionary of possible SHA-1 patterns (Rainbow Table) that use all the variants possible, we will see that this SHA-1 matches our pattern of 1,2,3,4 (or 0,1,2,3). The

companion site (Chapter 19), contains text file titled "*Gesture Dictionary_Rainbow table— Notepad.*" This was used to compare the SHA-1 value we created from our bytes of 00 01 02 03. Fig. 19.6 shows the SHA-1 value created using DigitalVolcano Hash Tool 1.1 followed by pasting the value using Ctrl+F keys into the dictionary. The matching values are highlighted in the bottom image of Figure 19.6.

	O DigitalVolcano Hash Tool 1.1
	File Options About
	Input File Hash Type
	Select File(p) Select file(p) or drag from Windows Explorer
	Last Hash
	a02a05b025b928c039cf1ae7e8ee04e7c190c0db Copy
	Results
	a02a05b025b928c039cf1ae7e8ee04e7c190c0db,C:\Users\John\Desktop\(^
/	
1	K
	Copy to Olpboard Clear
	· · · · · · · · · · · · · · · · · · ·
	Gesture Dictionary_Rainbow table - Notepad
	File Edit Format View Help
	Android OS gesture.key dictionary
	Gesture/Pattern sequence/Pattern SHA-1;
	descure/raccern sequence/raccern six-1,
	1234;00 01 02 03;A02A05B025B928C039CF1AE7E8EE04E7C190C0DB
	1235;00 01 02 04;6E36A9BFACF6C4637D64042B11CB78BFDDAF8BF3
	1236;00 01 02 05;101B2A675E9FB9546336D5B9EF70418B594184F4 1237;00 01 02 06;30F26B9825EA6F2EF6DBF6A88959774314D83F65
	1238;00 01 02 07;8C33B9D8BB8398DA8F19D9E946D589F32321B7BD
	1239;00 01 02 08;54EA40F7D20D6E7AF5A46A5205D048C721DF7D67
	1243;00 01 03 02;4D9B2A56701B8861A4030A64A734AC377DD77347
	1245;00 01 03 04;19E22522C33D2DFF774020CBBFF8D62CFA622493
	1246;00 01 03 05;9F3AB70FB4BFD980D46194CD69CCDC830053517D
	1247;00 01 03 06;DD3D13D4C56ABA79E8B5B8B18B4DA9B39F900A02 1248;00 01 03 07;7B77E822FDC554BBA832BBA1315754C7034B3B63
	1249;00 01 03 08;460351397D49BD510B8866B4D36C1AD456F29FC9
	1253;00 01 04 02;0F504B78F2DEF8E692A71D62D9692053AA282709
	1254;00 01 04 03;59A3556203C8F6C908D6C6BCE4C5E03F6BF343E3
	1256;00 01 04 05;C9398E8C8D89F82C9D48EEFC56742F81FCE73B68
	1257;00 01 04 06;2513AE43636AEEE1FDE429DC51FD848352309B8B 1258;00 01 04 07;8E9D7E8E6BC4E77529D93115D84F65D6C7008261
	1259;00 01 04 08;28ACB1A6B841EC908F4B53D46E49D32788E19968
	1263;00 01 05 02;2299C757D1463EFD2AD65F336F5820C681B749EC
	1264;00 01 05 03;E0326CB69A2650E46300D7BDDC9A17B8F8A9CF37
	1265;00 01 05 04;DC5F263A9400F8C16A3739A8C59B1EE01CA23658
	1267;00 01 05 06; B97AC6AC5CD5A24B972B15378F8CF6828F978926
	1268;00 01 05 07;D2BC144F8FA7F7BC77C9DB31AEBB705E202A4FC9 1269;00 01 05 08;E24F72761ADB7AEEB24B7D568C9299464073ADDA
	1273;00 01 06 02;A5D80B38ABD9240A8E4E6D78FBA114CA533665BA
	1274;00 01 06 03;033BFACD5E071194A58B7C5301D0FD8D885C7604
	1275;00 01 06 04;88CDD5377B01306EB1D7F378FDE9CC20A63CBD8E
	1276;00 01 06 05;951D28883DB5BE962C3E739CC8BE5D75C2F71596
	1278;00 01 06 07;8BAE6C7CAF2645E3E1182210C1E9E60310387A4A 1279;00 01 06 08;97C02DB1D0AEFF823734BC84E22BED274ED09F82
	12, 3, 00 01 00, 37 C0200100AEFF0237 540C042220E0274E003F02

Figure 19.6

DigitalVolcano Hash Tool and comparison with an Android Gesture dictionary (Rainbow Table).

This rainbow table can be utilized to decode any SHA-1 value that is in the gesture.key location. This is also how Physical Analyzer decodes the same entry. By understanding how the pattern is stored, as well as how to replicate and decode the same SHA-1 byte values, examiners should now have a better understanding of this security feature on Androids.

Chapter Summary Key Points

Simple (nonsmart style) phones typically store the user enabled security credentials on the phone. The values are usually easy to decode. As phones become more complex, so do the types of security features that are available to the user. Android users continue to employ both the gesture pattern and password. The user can create a password with various combinations of characters up to 16 entries. Android passwords use a combination of SHA-1 and MD5. The value is also salted with a random integer. If examiners are decoding a file that has not been rebuilt with databases, they can employ search techniques to locate the general area of the salt value for the password lock. If the partitions and subsequent databases have been rebuilt, the salt location is much easier to find.

The integer value of the salt will need to be converted to hexadecimal, and uppercase letters converted to lowercase, when examiners use older versions of Hashcat to brute force the password. Once this is completed, the SHA-1 or MD5 can be combined with the converted salt value. Specific line commands can be found in the help menu to assist with using Hashcat. The system used must have a specific type of video card and driver. The more cards that are used, the faster the brute force works.

The gesture pattern that users store on their Androids is stored as a SHA-1 value. This will be in the gesture.key location within the file system. Examiners can manually decode this value using a dictionary (Rainbow Table). To help better understand how the bytes of the gesture are stored, examiners can replicate the actual bytes, create a SHA-1 value, and manually locate the same value in a dictionary. This can help to better understand the process and assist during any explanations needed during the judicial process.

This chapter briefly touches on how two popular forms of Android security are stored, and how examiners may go about locating and potentially cracking the security. There are many more forms of security at the user's disposal. If you happen to own an Android and use either the gesture pattern or password security, hopefully you understand these concepts even more.

References

- [1] Hashcat, Advanced Password Recovery, January 6, 2017. https://hashcat.net/hashcat/.
- [2] What's in an Android Password?, December 31, 2012. http://linuxsleuthing.blogspot.com/2012/10/android-pinpassword-cracking-halloween.html.
- [3] Freeware Hex Editor XVI32, June 26, 2012. http://www.chmaas.handshake.de/delphi/freeware/xvi32/xvi32.htm.
- [4] Digital Volcano Software. https://www.digitalvolcano.co.uk/hash.html.