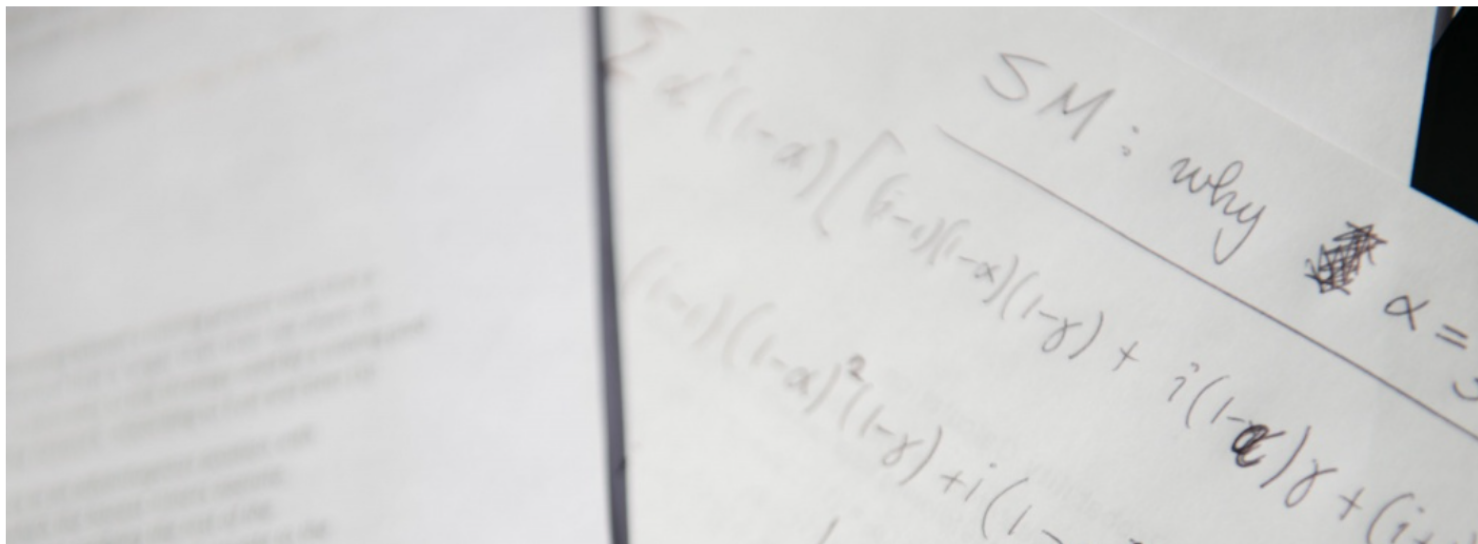


DR. CRAIG WRIGHT

ANALYSIS

JEAN-PAUL SARTRE, SIGNING AND SIGNIFICANCE

MAY 2, 2016



"If I sign myself Jean-Paul Sartre it is not the same thing as if I sign myself Jean-Paul Sartre, Nobel Prizewinner"

- Jean-Paul Sartre, 1964

I remember reading that quote many years ago, and I have carried it with me uncomfortably ever since. However, after many years, and having experienced the ebb and flow of life those years have brought, I think I am finally at peace with what he meant. If I sign *Craig Wright*, it is not the same as if I sign *Craig Wright, Satoshi*.

I think this is true, but in my heart I wish it wasn't.

```
IFdyawdodCwgaXQgaXMgmb90IHRoZSBzYW11IGFzIGlmIEkgc2lnbiBDcmFpZyBXcmInaHQsIFNh  
dG9zaGkuCgo=
```

I have been staring at my screen for hours, but I cannot summon the words to express the depth of my gratitude to those that have supported the bitcoin project from its inception – too many names to list. You have dedicated vast swathes of your time, committed your gifts, sacrificed relationships and REM sleep for years to an open source project that could have come to nothing. And yet still you fought. This incredible community's passion and intellect and perseverance has taken my small contribution and nurtured it, enhanced it, breathed life into it. You have given the world a great gift. Thank you.

Be assured, just as you have worked, I have not been idle during these many years. Since those early days, after distancing myself from the public persona that was Satoshi, I have poured every measure of myself into research. I have been silent, but I have not been absent. I have been engaged with an exceptional group and look forward to sharing our remarkable work when they are ready.

Satoshi is dead.

But this is only the beginning.

KEY VERIFICATION

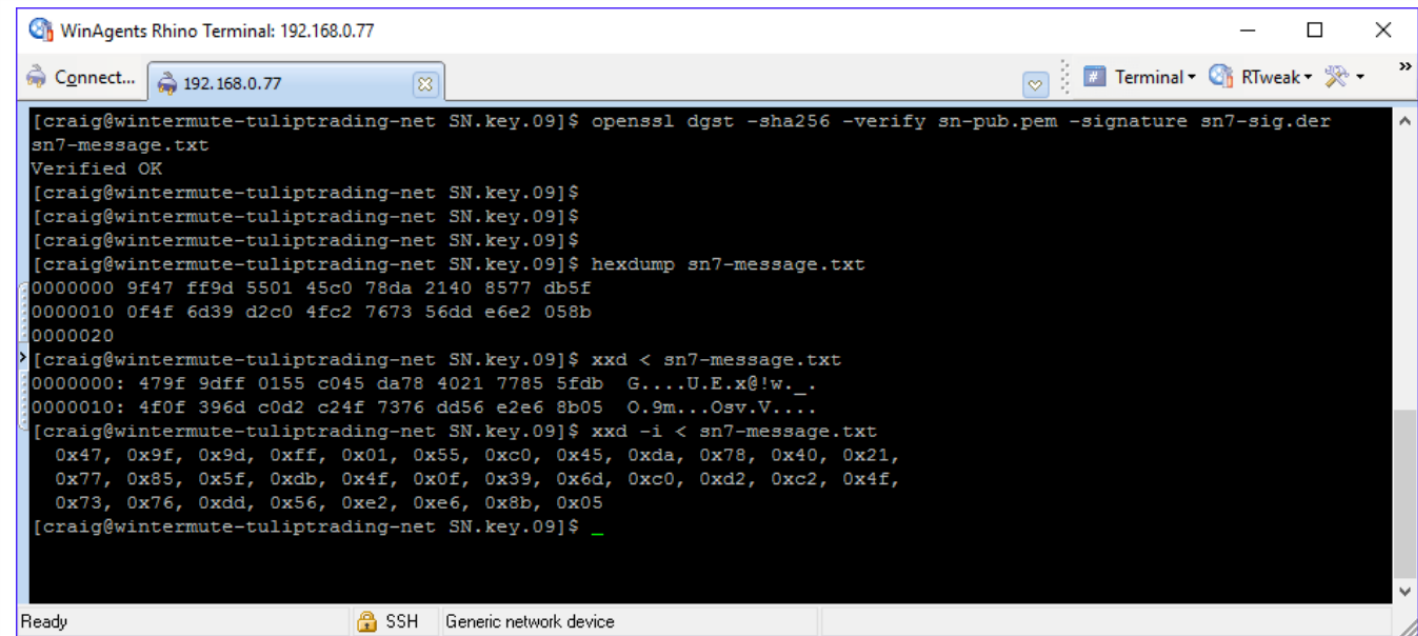
In the remainder of this post, I will explain the process of verifying a set of cryptographic keys.

To ensure that we can successfully sign and validate messages using the correct elliptic curve parameters in OpenSSL, it is necessary to ensure that the secp256k1 curve is loaded. This is not the default on Centos Linux. I will not detail this process here. I do point out that [RPMForge](#) maintains binaries that have already been patched. My recommendation would be to download both the source files from the OpenSSL website and the patch, if, like me you're running Centos.

I will also point the reader to the following websites for some preliminary reading:

- https://wiki.openssl.org/index.php/Command_Line_Elliptic_Curve_Operations
- <http://www.secg.org/sec2-v2.pdf>
- <https://www.openssl.org/>
- <https://www.bfccomputing.com/bitcoin-and-curve-secp256k1-on-fedora/>

The first stage of this exercise will be to explain hash functions. In the figure below we're displaying a file called "sn7-message.txt".



```
WinAgents Rhino Terminal: 192.168.0.77
Connect... 192.168.0.77
[craig@wintermute-tuliptrading-net SN.key.09]$ openssl dgst -sha256 -verify sn-pub.pem -signature sn7-sig.der
sn7-message.txt
Verified OK
[craig@wintermute-tuliptrading-net SN.key.09]$
[craig@wintermute-tuliptrading-net SN.key.09]$
[craig@wintermute-tuliptrading-net SN.key.09]$ hexdump sn7-message.txt
00000000 9f47 ff9d 5501 45c0 78da 2140 8577 db5f
00000010 0f4f 6d39 d2c0 4fc2 7673 56dd e6e2 058b
00000020
> [craig@wintermute-tuliptrading-net SN.key.09]$ xxd < sn7-message.txt
00000000: 479f 9dff 0155 c045 da78 4021 7785 5fdb G...U.E.x@!w._.
00000010: 4f0f 396d c0d2 c24f 7376 dd56 e2e6 8b05 O.9m...Osv.V...
[craig@wintermute-tuliptrading-net SN.key.09]$ xxd -i < sn7-message.txt
 0x47, 0x9f, 0x9d, 0xff, 0x01, 0x55, 0xc0, 0x45, 0xda, 0x78, 0x40, 0x21,
 0x77, 0x85, 0x5f, 0xdb, 0x4f, 0x0f, 0x39, 0x6d, 0xc0, 0xd2, 0xc2, 0x4f,
 0x73, 0x76, 0xdd, 0x56, 0xe2, 0xe6, 0x8b, 0x05
[craig@wintermute-tuliptrading-net SN.key.09]$
```

Ready SSH Generic network device

Script fragment

The series of hexadecimal values displayed in the figure above represents the SHA256 hash of an input value. A good hash algorithm will produce a large string of values that cannot be determined in advance. The amount of information and possible permutations always exceeds the range of imitations that can be output from any hash function and as a result, collisions will always exist. What makes a hash function such as SHA256 useful and considered "secure" is that it is infeasible given the current state of technology to determine and find a set of input values to the hash function that collides with the same value that is returned as output.

The SHA256 algorithm provides for a maximum message size of $(2^{128} - 1)$ bits of information whilst returning 32 bytes or 256 bits as an output value. The number of possible messages that can be input into the SHA256 hash function totals $(2^{128} - 1)!$ possible input values ranging in size from 0 bits through to the maximal acceptable range that we noted above.

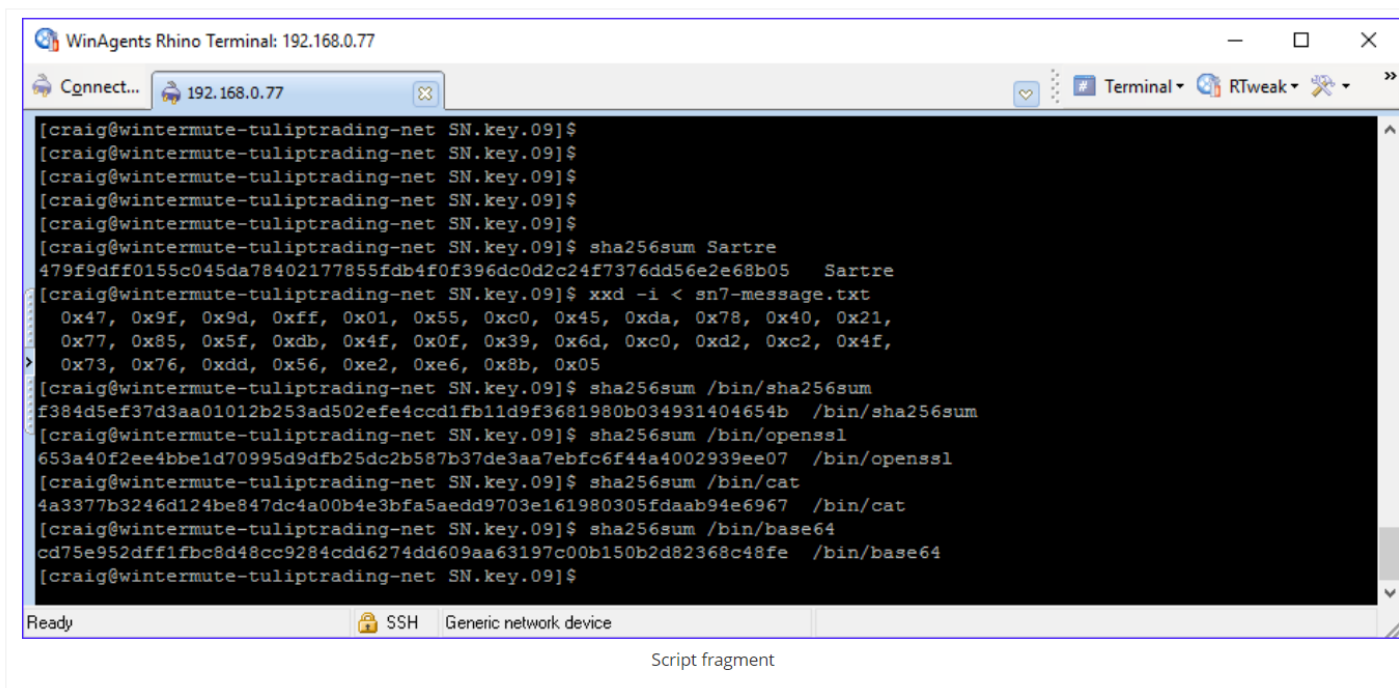
In determining the possible range of collisions that would be available on average, we have a binomial coefficient $\binom{n}{k}$ that determines the permutations through a process known as combinatorics [1].

I will leave it to a later post to detail the mathematics associated with collision detection. It is important to note though that there are an incredibly large number of colliding values associated with each hash but that the probability of finding two

colliding values or determining them in advance is infinitesimally small. Next week, I will follow-up with a post based on combinatorics and probability theory demonstrating the likelihood of finding collisions for “secure” hashing algorithms.

HASHING

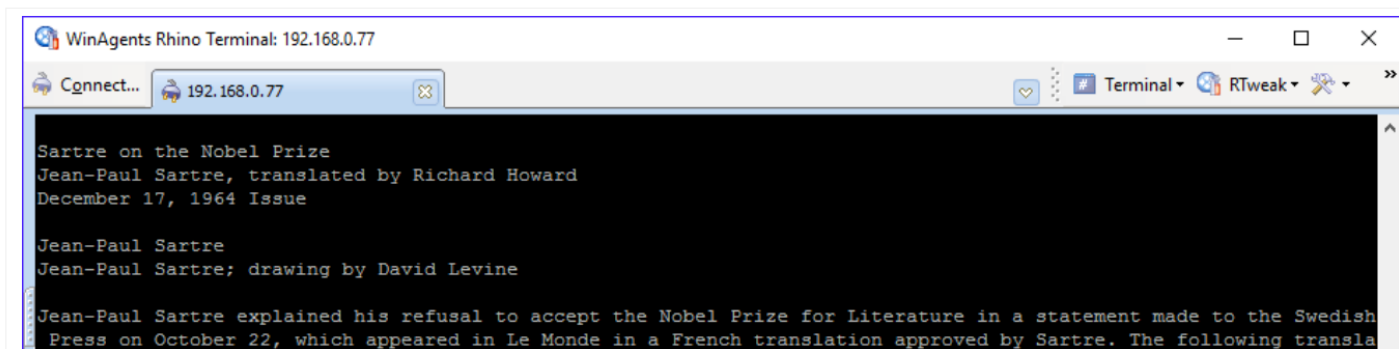
Hash functions are relatively simple and can be done by hand. This of course belies the complexity that is required to reverse them. A good hash function is simple to use and yet is infeasible to reverse. In the figure below we have run the Linux hash routine “sha256sum”. This simple program will return a unique value that corresponds to a set and fixed input.



```
WinAgents Rhino Terminal: 192.168.0.77
Connect... 192.168.0.77
[craig@wintermute-tuliptrading-net SN.key.09]$
[craig@wintermute-tuliptrading-net SN.key.09]$
[craig@wintermute-tuliptrading-net SN.key.09]$
[craig@wintermute-tuliptrading-net SN.key.09]$
[craig@wintermute-tuliptrading-net SN.key.09]$
[craig@wintermute-tuliptrading-net SN.key.09]$ sha256sum Sartre
479f9dff0155c045da78402177855fdb4f0f396dc0d2c24f7376dd56e2e68b05 Sartre
[craig@wintermute-tuliptrading-net SN.key.09]$ xxd -i < sn7-message.txt
 0x47, 0x9f, 0x9d, 0xff, 0x01, 0x55, 0xc0, 0x45, 0xda, 0x78, 0x40, 0x21,
 0x77, 0x85, 0x5f, 0xdb, 0x4f, 0x0f, 0x39, 0x6d, 0xc0, 0xd2, 0xc2, 0x4f,
 0x73, 0x76, 0xdd, 0x56, 0xe2, 0xe6, 0x8b, 0x05
[craig@wintermute-tuliptrading-net SN.key.09]$ sha256sum /bin/sha256sum
f384d5ef37d3aa01012b253ad502efe4ccd1fb11d9f3681980b034931404654b /bin/sha256sum
[craig@wintermute-tuliptrading-net SN.key.09]$ sha256sum /bin/openssl
653a40f2ee4bbe1d70995d9dfb25dc2b587b37de3aa7ebfc6f44a4002939ee07 /bin/openssl
[craig@wintermute-tuliptrading-net SN.key.09]$ sha256sum /bin/cat
4a3377b3246d124be847dc4a00b4e3bfa5aedd9703e161980305fdaab94e6967 /bin/cat
[craig@wintermute-tuliptrading-net SN.key.09]$ sha256sum /bin/base64
cd75e952dff1fbc8d48cc9284cdd6274dd609aa63197c00b150b2d82368c48fe /bin/base64
[craig@wintermute-tuliptrading-net SN.key.09]$

Ready SSH Generic network device
Script fragment
```

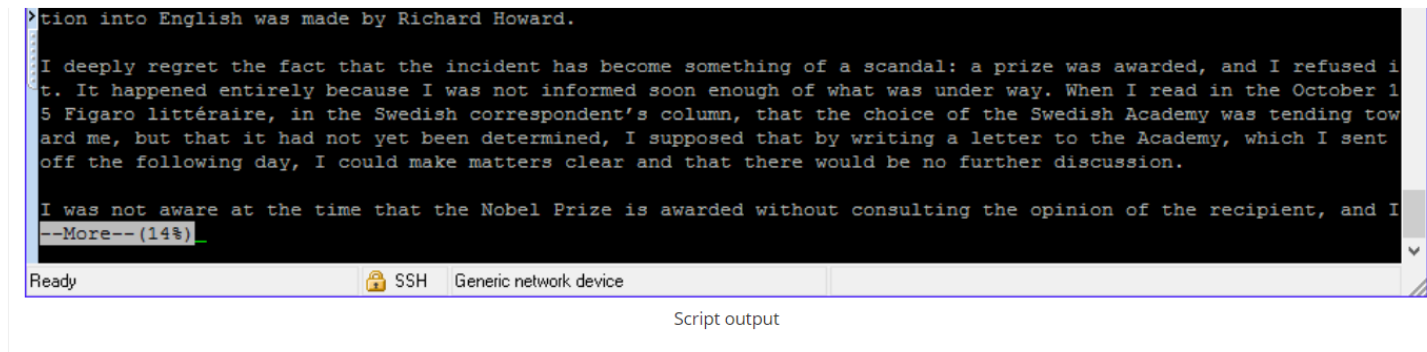
In the figure above, we have run this on several files including one that we are using for this OpenSSL signature exercise. The particular file that we will be using is one that we have called Sartre. The contents of this file have been displayed in the figure below.



```
WinAgents Rhino Terminal: 192.168.0.77
Connect... 192.168.0.77
Sartre on the Nobel Prize
Jean-Paul Sartre, translated by Richard Howard
December 17, 1964 Issue

Jean-Paul Sartre
Jean-Paul Sartre; drawing by David Levine

Jean-Paul Sartre explained his refusal to accept the Nobel Prize for Literature in a statement made to the Swedish Press on October 22, which appeared in Le Monde in a French translation approved by Sartre. The following transla
```

A screenshot of a terminal window with a black background and white text. The text is a paragraph in English, starting with "tion into English was made by Richard Howard." and ending with "I was not aware at the time that the Nobel Prize is awarded without consulting the opinion of the recipient, and I". Below the text, there is a prompt "--More-- (14%)". The terminal window has a title bar that says "Ready" and "SSH Generic network device". Below the terminal window, the text "Script output" is visible.

```
>tion into English was made by Richard Howard.  
  
I deeply regret the fact that the incident has become something of a scandal: a prize was awarded, and I refused i  
t. It happened entirely because I was not informed soon enough of what was under way. When I read in the October 1  
5 Figaro littéraire, in the Swedish correspondent's column, that the choice of the Swedish Academy was tending tow  
ard me, but that it had not yet been determined, I supposed that by writing a letter to the Academy, which I sent  
off the following day, I could make matters clear and that there would be no further discussion.  
  
I was not aware at the time that the Nobel Prize is awarded without consulting the opinion of the recipient, and I  
--More-- (14%)
```

Digital signature algorithms sign the hash of the message. It is possible to sign the message itself but in signing the hash it is possible to ensure the integrity of the message and validate that the message has not changed. If even a single space or "." was to be altered, the hash will be radically different to the value returned initially.

In order write this value and save it to a file, we can use the Linux command, `xxd`. This will write the ASCII values into a hexadecimal binary file. In the command below we would be writing a string of zeros into a file called "file.name".

```
echo '000...000' | xxd -r -p > file.name
```

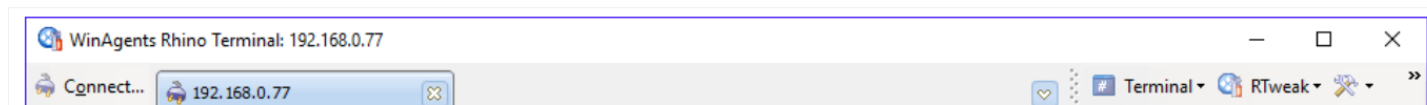
In doing this, we can change the string we received as output from the hashing algorithm into a hex encoded file. This will be the message we can sign and verify. It is important to validate the string of numbers that you are putting into the echo command above. If a single digit has been typed incorrectly then the message will not verify.

PUBLIC KEYS

In order to verify a digitally signed message we need number of components. These include:

- The algorithm,
- the public key of the signing party that we wish to verify,
- the message that has been signed, and
- the digital signature file.

The first part of this, the algorithm is obtained through the installation of OpenSSL with the incorporation of the `secp256k1` curve patch. In the step above we covered the creation of a hashed message. In the next section we will cover the use of ECDSA public keys.



```
[craig@wintermute-tuliptrading-net SN.key.09]$
[craig@wintermute-tuliptrading-net SN.key.09]$
[craig@wintermute-tuliptrading-net SN.key.09]$
[craig@wintermute-tuliptrading-net SN.key.09]$ cat signature.der
MEUCIQDBKn1Uly8m0UyzETObUSL4wYdBfd4ejvtoQEVcNCIK4AIgZmMsXNQHwo6KDd2Tu6euE11
3VTC3ihl6XUlhC+fm4=
[craig@wintermute-tuliptrading-net SN.key.09]$ openssl ec -in sn-pub.pem -pubin -text -noout
read EC key
Private-Key: (256 bit)
pub:
04:11:db:93:e1:dc:db:8a:01:6b:49:84:0f:8c:53:
bc:1e:b6:8a:38:2e:97:b1:48:2e:ca:d7:b1:48:a6:
90:9a:5c:b2:e0:ea:dd:fb:84:cc:f9:74:44:64:f8:
2e:16:0b:fa:9b:8b:64:f9:d4:c0:3f:99:9b:86:43:
f6:56:b4:12:a3
ASN1 OID: secp256k1
[craig@wintermute-tuliptrading-net SN.key.09]$ xxd -i sn-pub.pem
unsigned char sn_pub_pem[] = {
0x2d, 0x2d, 0x2d, 0x2d, 0x2d, 0x42, 0x45, 0x47, 0x49, 0x4e, 0x20, 0x50,
0x55, 0x42, 0x4c, 0x49, 0x43, 0x20, 0x4b, 0x45, 0x59, 0x2d, 0x2d, 0x2d,
0x2d, 0x2d, 0x0a, 0x4d, 0x46, 0x59, 0x77, 0x45, 0x41, 0x59, 0x48, 0x4b,
0x6f, 0x5a, 0x49, 0x7a, 0x6a, 0x30, 0x43, 0x41, 0x51, 0x59, 0x46, 0x4b,
0x34, 0x45, 0x45, 0x41, 0x41, 0x6f, 0x44, 0x51, 0x67, 0x41, 0x45, 0x45,
0x64, 0x75, 0x54, 0x34, 0x64, 0x7a, 0x62, 0x69, 0x67, 0x46, 0x72, 0x53,
0x59, 0x51, 0x50, 0x6a, 0x46, 0x4f, 0x38, 0x48, 0x72, 0x61, 0x4b, 0x4f,
0x43, 0x36, 0x58, 0x73, 0x55, 0x67, 0x75, 0x79, 0x74, 0x65, 0x78, 0x53,
0x4b, 0x61, 0x51, 0x6d, 0x6c, 0x79, 0x0a, 0x79, 0x34, 0x4f, 0x72, 0x64,
0x2b, 0x34, 0x54, 0x4d, 0x2b, 0x58, 0x52, 0x45, 0x5a, 0x50, 0x67, 0x75,
0x46, 0x67, 0x76, 0x36, 0x6d, 0x34, 0x74, 0x6b, 0x2b, 0x64, 0x54, 0x41,
0x50, 0x35, 0x6d, 0x62, 0x68, 0x6b, 0x50, 0x32, 0x56, 0x72, 0x51, 0x53,
0x6f, 0x77, 0x3d, 0x3d, 0x0a, 0x2d, 0x2d, 0x2d, 0x2d, 0x45, 0x4e,
0x44, 0x20, 0x50, 0x55, 0x42, 0x4c, 0x49, 0x43, 0x20, 0x4b, 0x45, 0x59,
0x2d, 0x2d, 0x2d, 0x2d, 0x2d, 0x0a
};
unsigned int sn_pub_pem_len = 174;
[craig@wintermute-tuliptrading-net SN.key.09]$ _
```

Script fragment

For this exercise I am using a public-private key pair that is saved as a PEM file in OpenSSL. David Derosa has written an excellent page defining the creation of an elliptic curve key pair in OpenSSL. In the figure above you can see the particular PEM format public key that is associated with the key pair used in signing the message in this exercise. A thorough reading of David's page will provide all of the information for the reader detailing how a private key pair used in bitcoin transaction can be formatted as a PEM file. This page details the creation of a new private key and not how an existing private key can be imported into OpenSSL. I shall cover this additional process and demonstrate how an existing private key pair based on elliptic curve cryptography can be imported into a ASN.1 format for use with OpenSSL directly.

The command to export our public key is given below.

```
openssl ec -in sn-pub.pem -pubin -text -noout
0411db93e1dccb8a016b49840f8c53
bc1eb68a382e97b1482ecad7b148a6
909a5cb2e0eaddfb84ccf9744464f8
2e160bfa9b8b64f9d4c03f999b8643
f656b412a3
```

The string returned is the public key value used by programs including bitcoin for the verification and addressing of the signing function.

Casascius has developed a nifty tool that will help you decode this public key and return the associated bitcoin address that it maps to. We have a blog on this site that will help you understand the technical aspects of how bitcoin addresses derived from the public and private keys. Several [online tools](#) are also available that can calculate the bitcoin address from the public key.

SIGNING

The process of digitally signing a message using OpenSSL requires that the party signing the message has access to the private key. I will document and cover this process further in a later post. In recent sessions, I have used a total of 10 private keys are associated with bitcoin addresses. These were loaded into [Electrum](#), an SPV wallet. In one of the exercises, I signed messages that I will not detail on this post for a number of individuals. These were not messages that I personally selected, but rather ones that other people had selected. In some instances, we ensure the integrity of the process by downloading a new version of the electrum program, installing it on a fresh laptop that has just been unboxed having been purchased that afternoon and validating the signed messages on the new machine.

The version of electrum that I run is on Centos Linux v7 and runs via Python. For the exercise I noted above we used Windows 7 and Windows 10 on different occurrences.

SIGNATURE VERIFICATION

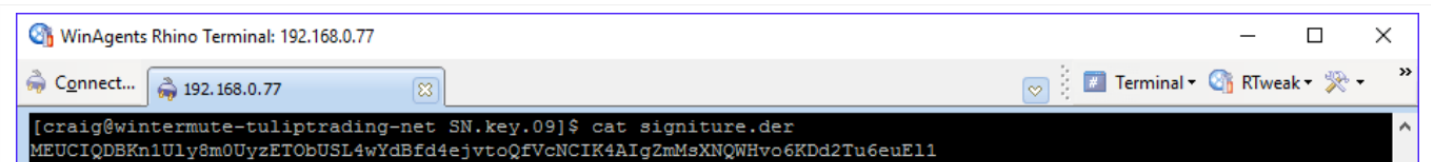
The final component that we need to cover is the signature itself. We will be using the following command to convert our base64 format signature into a file format that can be loaded into OpenSSL.

```
>> base64 --decode signature > sig.asn1 & openssl dgst -verify sn-pub.pem -signature sig.asn1 sn7-message.txt
```

The signature file we will be verifying contains the following data.


```
----- Signature File -----  
MEUCIQDBKn1Uly8m0UyzETObUSL4wYdBfd4ejvtoQfVcNCIK4AIgZmMsXNQWHvo6KDD2Tu6euE11  
3VTC3ih16XU1hcU+fM4=  
----- End Signature -----
```

In the figure below we display the signature file as it is stored on the computer that was used for this process and we see the result of the verification exercise. In saving this file, you could cut-and-paste the encoded signature and insert it into a saved file using an editor program such as vim. Not that I'm looking at getting into a holy war over the choice of editing programs.



```
WinAgents Rhino Terminal: 192.168.0.77  
Connect... 192.168.0.77  
[craig@wintermute-tuliptrading-net SN.key.09]$ cat signature.der  
MEUCIQDBKn1Uly8m0UyzETObUSL4wYdBfd4ejvtoQfVcNCIK4AIgZmMsXNQWHvo6KDD2Tu6euE11
```

```
3VTC3ih16XU1hcU+fM4=
[craig@wintermute-tuliptrading-net SN.key.09]$
[craig@wintermute-tuliptrading-net SN.key.09]$ base64 --decode signature.der > sig.asn1 & openssl dgst -verify sn-
pub.pem -signature sig.asn1 sn7-message.txt
[1] 21359
Verified OK
[1]+ Done base64 --decode signature.der > sig.asn1
[craig@wintermute-tuliptrading-net SN.key.09]$
> [craig@wintermute-tuliptrading-net SN.key.09]$
[craig@wintermute-tuliptrading-net SN.key.09]$ _
```

Ready  SSH Generic network device

Script fragment

There are two possible outputs from this process that concern us. OpenSSL will either return as “Verified OK” where we have validly verified the signature. All of the information that is required to import the public key, the message and the message signature used in this post is available on this post.

I could have simply signed a message in electrum as I did in private sessions. Loading such a message would have been far simpler. I am known for a long history of “being difficult” and disliking being told what “I need to do”. The consequence of all of this is that I will not make it simple.

SOME SCRIPTS

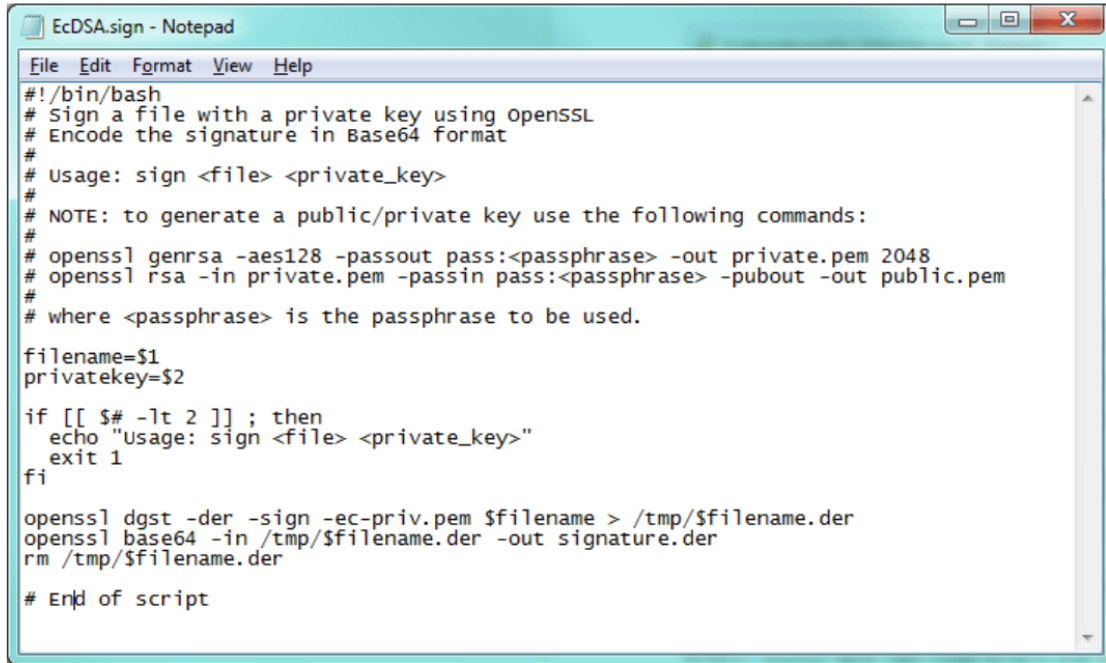
In order to simplify this process, I have included two shell scripts. For variations on scripts like these, please visit a site such as the one hosted by Enrico Zimuel. This site is not particularly focused on elliptic curve cryptography but it is not too difficult to update his code for the use on a bitcoin based system.

SIGNING

For you to try and test this at your leisure I have included the signing script below. To use this script, the input consists of the variable <file> which signifies the file that you desire to sign using a selected <private_key> under your control. In this command, the <private_key> variable represents the file containing the private key to be used in signing the message and which will output the signature.

```
EcDSA.Sign.sh <file> <private_key>
```

The output from this shell script consists of the signature saved as a Base64 encoded file. This will be saved to your hard drive or other location using Base64 format as a file named <signature.der>.



```
EcDSA.sign - Notepad
File Edit Format View Help
#!/bin/bash
# Sign a file with a private key using openssl
# Encode the signature in Base64 format
#
# Usage: sign <file> <private_key>
#
# NOTE: to generate a public/private key use the following commands:
#
# openssl genrsa -aes128 -passout pass:<passphrase> -out private.pem 2048
# openssl rsa -in private.pem -passin pass:<passphrase> -pubout -out public.pem
#
# where <passphrase> is the passphrase to be used.
filename=$1
privatekey=$2
if [[ $# -lt 2 ]] ; then
  echo "usage: sign <file> <private_key>"
  exit 1
fi
openssl dgst -der -sign -ec-priv.pem $filename > /tmp/$filename.der
openssl base64 -in /tmp/$filename.der -out signature.der
rm /tmp/$filename.der
# End of script
```

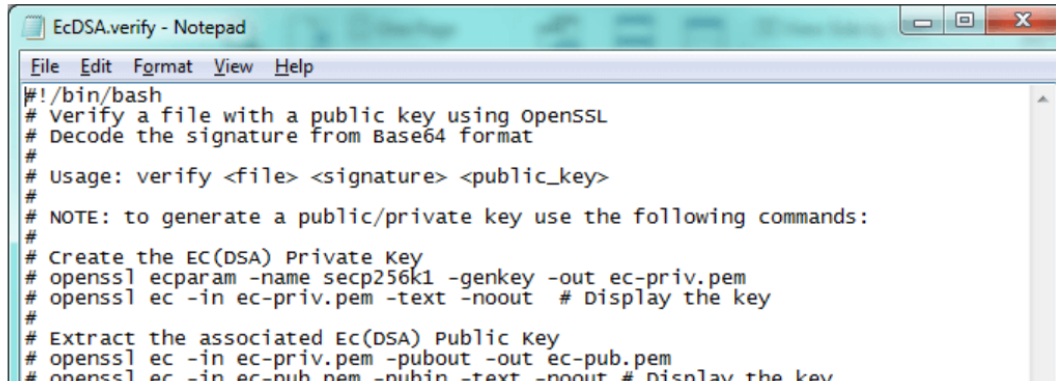
EcDSA.sign.sh

VERIFICATION

We can use a similar process to verify the signature we have created using the script that I have included below.

```
EcDSA.Verify.sh <file> <signature> <public_key>
```

In this commandline, the variable <file> is used to signify the name of the file we seek to verify. The variable <signature> represents the file where we have saved the signature (and coded using Base64), and the final variable, <public_key> contains the PEM formatted public key. We use these files together and if they are valid and correct they will allow us to successfully to verify the digital signature.



```
EcDSA.verify - Notepad
File Edit Format View Help
#!/bin/bash
# verify a file with a public key using openssl
# Decode the signature from Base64 format
#
# Usage: verify <file> <signature> <public_key>
#
# NOTE: to generate a public/private key use the following commands:
#
# Create the EC(DSA) Private Key
# openssl ecparam -name secp256k1 -genkey -out ec-priv.pem
# openssl ec -in ec-priv.pem -text -noout # Display the key
#
# Extract the associated Ec(DSA) Public Key
# openssl ec -in ec-priv.pem -pubout -out ec-pub.pem
# openssl ec -in ec-pub.pem -pubin -text -noout # Display the key
```

```
#
# where <passphrase> is the passphrase to be used.

filename=$1
signature=$2
publickey=$3

if [[ $# -lt 3 ]] ; then
  echo "usage: verify <file> <signature> <public_key>"
  exit 1
fi

base64 --decode $signature > /tmp/$filename.sig
# the optional flag -sha256 is not needed for secp256k1 as this is the default
openssl dgst -verify $publickey -signature /tmp/$filename.sig $filename
# Clean up the system.
rm /tmp/$filename.sig

# End of script
```

EcDSA.verify.sh

CHOICES ON FORMATTING

The signature format used within bitcoin is based on DER encoding. Other methods have been applied in the original code has changed significantly in the last seven years. The choice of DER encoding for the signatures and other information was based on a desire to ensure that information could be shared between incompatible systems. It is not the most efficient means of storing information but it does allow for disparate systems to communicate efficiently.

Like many open source projects, OpenSSL is poorly documented in many areas. bitcoin addressing and the storage of key pairs could have been far more efficient and the code has been updated to ensure that this is now the case. But like every new system it is far better to have something that is working on something that is not available but is aiming at perfection.

Security is always a risk function and not an absolute.

REFERENCES

- [1] Lovasz, Laszlo (1979) "Combinatorial Problems and Exercises" North Holland Publishing Co. Amsterdam

