Australian Government

Cyber Security Challenge Australia 2014
www.cyberchallenge.com.au

CySCA2014 Crypto Writeup

Background: Fortcerts have created a number of programs involving cryptographic algorithms. They have self certified these programs and want a second opinion on the security of the cipher implementations.

Crypto 1 - Standard Galactic Alphabet

Question: Perform a white box evaluation of the custom encryption used in Fortcerts "Slightly Secure Shell" program. Identify any vulnerabilities in their implementation and demonstrate that they can be exploited to gain confidential information. The server is running at 172.16.1.20:12433

Designed Solution: Players send an echo command before their actual command to get the cipher key and decipher the output. They then use multiple commands to find the key file and view its contents.

Write Up:
We start by reading the source code provided with the challenge. We determine that the program takes user input until it finds a newline or carriage return, it then executes the command using bash, enciphers the output from the command using a mono-alphabetic substitution and returns the output back to the user. After this the cipher key is reset.

Because it is piping commands to bash we can perform multiple commands while using one cipher key. Perhaps we can use this fact to recover the key and decrypt the output.

We connect to the server and send some commands to test our assumptions that we can run multiple commands and that the key is only reset after the command is run.

```
#> nc 172.16.1.20 12433
You have connected to the Slightly Secure Shell server of Fortress Certifications.
#>echo AAAA
echo AAAARunning command: 'echo AAAA'
6666
Key reset
#>echo AAAA;echo AAAA
echo AAAA;echo AAAARunning command: 'echo AAAA;echo AAAA'
zzzz
zzzz
Key reset
#>echo AAAA;echo BBBB; echo ABAB
echo AAAA;echo BBBB; echo ABABRunning command: 'echo AAAA;echo BBBB; echo ABAB'
8888
5555
8585
Key reset
```

Seeing that the A's and B's are always being replaced by the same character we can see it is a mono-alphabetic cipher, and we can see that the key is being reset after all the command output is displayed.

We can pair two bash commands to run arbitrary commands, in the first we echo the plaintext string and in the second we run our actual command. The first line of output will be the ciphertext alphabet and the rest will be the second command output.

We test this idea by echoing a simple plaintext alphabet and echoing a test string. We can then use the ciphertext to decode the output.

```
#>echo abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ; echo TestString
echo abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ; echo TestStringRunning
command: 'echo abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ; echo
TestString'
_u;pSw"y|r^QFJ <Lk\+{KnEX$#%=UDb/6[HlBOaPRA54(-!'xq: <<< Ciphertext alphabet
(S\+4+k|J" <<< Test string
```

We then use our the ciphertext alphabet to decode the test string.

```
 Plain Alphabet: abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ
Cipher Alphabet: _u;pSw"y|r^QFJ <Lk\+{KnEX$#%=UDb/6[HlBOaPRA54(-!'xq:
Cipher Test string: (S\+4+k|J"
Plain  Test String: TestString
```

Now that we know this method works we will locate the flag using the ls command, and then display it using the cat command.

```
#> echo abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ1234567890.;ls -1
```

```
echo abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ1234567890.;ls -1Running
command: 'echo abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ1234567890.;ls
-1'
 qaF)-:"XzJ^#Ehp2H_*]iDMUP\@8`rNdIm6AW3&5e%0}[Rk/VYS>{tl<;KfG4n <<Key
qXE       << bin
F)i       << dev
)*a       << etc
-^ :n*M*  << flag.txt
^Xq        << lib
Key reset
#> echo abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ1234567890.; cat
flag.txt
echo abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ1234567890.; cat
flag.txtRunning command: 'echo
abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ1234567890.; cat flag.txt'
q^0sE`"{G3KSo& @):nUZ=;+[BM?Lx*]CwIdmOg#2,VX-!Plav'e\9N4/_F(Wf7
Lq=Gq:?qKEs-{qoEW__    << CaviarBakedShame966
```

After running these commands and decoding the responses we get the flag for this question **CaviarBakedShame966**

Crypto 2 - Compression Session

Question: Perform a white box evaluation of the Fortcerts highly secure key generation server. Identify and exploit any vulnerabilities in the implementation that will lead to a disclosure of secret data. The server is running at 172.16.1.20:9999

Designed Solution: Players should identify the CRIME based attack here. Players then create a script that uses information leaked due to using compression before encryption with a stream cipher to leak information about the the secret key.

Write Up:
We start by analysing the source snippet provided with the challenge, we see that the server is basically just taking our input, appending it to the secret, compressing it, encrypting it and then returning the encrypted data as a hex encoded string. It is using AES in CTR mode which means AES is acting as a stream cipher rather than a block cipher.

We then connect to the given remote service using netcat. We are presented with a banner identifying this as a key server and that any unauthorised access will be treated as CRIME. We find the capitalisation of CRIME unusual, additionally the server does not provide any prompts for the user to interact with, nor does it provide any help functionality.

```
#> nc 172.16.1.20 9999
       Welcome to the Keygen server.
       ==============================
[+]All access is monitored and any unauthorised access will be treated as CRIME
```

To test that the assessment of the code functionality we made when looking at the source code snippet is valid, we send the server a number of inputs. From these we can see that the key is not being reset between each submission however the counter used in CTR mode is increasing so the encrypted data differs. We can also see that the data length changes by small amounts, showing that is using a stream cipher rather than a block cipher.

```
AAAA
0a7a99948f447f19b22d7a9a74e25d591b8ff864fa0d3b80804f18b2798219cdb4429e39c8bd6594bc
50f640432fa7db9368db69ce769d2fe4ca30a1bc728c4d0ccdf701b8ae79000db82220
AAAA
ae387373a2131e25e26793aae4d3d1e61a74680c0c654dcef76ec0c27667bf861bd87d272f8d70e03f
b55a515be355f0030bae31beef06903ee6da654a96dbe41d134bea66d4993fbb414512
AAAA
7ff3513568e314e954bb872b604d3d9518fb64382fc6129d80465f27f99ab53a0300aea122627cc2fa
2d5d06600626079c20406f51fe9dcd9a680a9c9041c421308ddc322aac3b2f7dc9f253
AAAABBBB
b1217dacfc0e7927f3519baedb290a14b791b3b97c821158e84e956d071c887f97afd85213476712ff
da16fab035d54a72ee8f8608f94557e7e8866480aa5252eb1ed533c8d006c647824fc95123782b
AAAAAAAA
```

```
a3c7b8ae0f2628e126914284f62db7f60122daa23fc39cfc7f3cbd8999c2f979a3f8a214ebd225c48e
5ec5b303a672a45450ef13c00c110758e14d3f2981b95356c537f678357340be941877
```

We notice from the output that eight A's encrypt the the same length as four A's. This seems unusual until we realize that the eight A's will generally compress to the same length as four A's. The compressed data is encrypted using a stream cipher so the length of the encrypted data is equal to the length of the compressed data. We can use the length of the compressed data to leak information about the data being compressed. This is a very similar principle to the CRIME attack. Do you see why the capitalization was there now ;)

After looking at the supplied code again we determine the layout of the compressed data, if we assume the flag was "TestFlag" and we supplied the server ABCD the compressed data would be "Key:TestFlagABCD". This data wouldn't compress very well and we would have a longer response string. However, if we entered "Key:" as our input, the data to be compressed would be "Key:TestFlagKey:". When the algorithm compresses this data it would notice the two instances of "Key:" and compress it more efficiently giving us a shorter response.

To test our assumption we send the server "Key:". Each time we sent this we got different response content, however the length of the responses did not change.
We then sent the server "Key:A" and the length of the response increased. Sending "Key:B" and "Key:C" to the server gave us the same response length as "Key:A". When we sent the server "Key:D" the response was shorter, with the same length as when we sent "Key:".

```
Key:
8bec8291d51787058bfe80c1303c3024f60b84812b4ec6a0d2a818dc74fc340bc45d99f89e29fbd7ee
c47f9091ebe518bd85b6e3d5c583719ee438ae4d597772afc9506aec2214865d9750ca
Key:A
964c86d2d100faaee0a920dee7716a29493cd3e11e7a33e1b0cfbe866e01f53208cab473a3eebc8607
dbf2ab25d197b6f6684c70f9a51b15debc4f282edf048cfb57c47b94e7b147b0f08fa4f27c
Key:B
85deee163f4e0982f67688fb0cdb7c735ccbe8f276796aadfdb7585a0528e5af9e107f5901637fdbc2
9c4622d821a91f4a4da9302115f601aeb97a120ffb86350797ed1e801a7e8c9f00f97e28dd
Key:C
af147c047eb3d0e84472cc95670919617ae4d10623b990ced591f074bae9d85f58bce66491ccc2a9c5
6ff9e438e78ce02f9c4d61086a5f52f16806a2eddc6d7daca5498c15ac854018a53491ab8d
Key:D
dd008730d0524e3af6332e4c69e69d7428968b7b21d32ebef9e35d8b908026b69023b09e2a9f978721
017b254b0b1b08ec7cd460a681e404eedb027c88e843f288ba5a15d276208aa86792b0
```

From this we could determine that the flag probably started with a capital D. Knowing the weakness we then wrote a Python script that would connect to the server and brute force each character of the flag one after another.

```
import socket
import string
```

```
known_part_of_the_key = "Key:"

def Checker(s, bs):
  global known_part_of_the_key
  strings = string.letters + string.digits + "%/+=:"
  while 1:
       for ch in strings:
       s.send(known_part_of_the_key + ch)
       resp = s.recv(2048)
       if len(resp) == bs:
       print known_part_of_the_key
       known_part_of_the_key = known_part_of_the_key + ch

host = "172.16.1.20"
port = 9999
sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
sock.connect((host, port))
banner = sock.recv(1024) #Receive the banner
print banner
sock.send(known_part_of_the_key)
resp = sock.recv(2048)
base_size = len(resp)
Checker(sock, base_size)
```

After running the script we get the output which includes the flag **DrizzleVerandaFinger576**

```
#> python soln.py
 Welcome to the Keygen server.
 ==============================
[+]All access is monitored and any unauthorised access will be treated as CRIME

Key:
Key:D
Key:Dr
Key:Dri
Key:Driz
****SNIP****
Key:DrizzleVerandaFinger576
****SNIP****
```

Crypto 3 - Chop Suey

Question: Senior staff at Fortcerts have expressed objections to the use of white box evaluation methodology with the argument "real attackers won't have source code access". Perform a black box evaluation of the Fortcerts very secure encryption service. Diagnose and identify any crypto vulnerabilities in the service that can be used to recover encrypted data. The very secure encryption service is running at 172.16.1.20:1337

Designed Solution: Players need to analyse the server to determine that it is using a stream based cipher algorithm. Once they have done this they need to determine that the IV's are being reused leading to repeated keystreams. Players then build a database of keystreams and use a repeated keystream to decrypt the flag.

Write Up:
Because no files are provided for this question we start by connecting to the encryption service to see if we can understand its functionality. We are greeted with a message saying Key Reset! and then a banner telling us that the program uses very secure encryption. Additionally, it lists two commands and tells us the format of the output.

```
#> nc 172.16.1.20 1337
Key Reset!
       Welcome to the FortCerts Certified Data Encryption Service
              This program uses very secure encryption
Commands:
E - Encrypt specified data
D - Dump service stored data
Output is in format <IV>:<Encrypted Data>
```

We try entering both the commands listed when we first connect to try determine their functionality. When we type E by itself we get an invalid use of encrypt, it seems like this command requires a value in addition to the E character. When we use the D command we get a string that seems encrypted which would match the help blurb next to the D command, additionally we get a message saying Key Reset!.

```
E
Invalid use of encrypt. Usage E,<valuetoencrypt>
D
226e:01ef3044bac49aed75821296a10c060f0d0225386936
Key Reset!
```

We try to determine if there is any undocumented commands by passing in a to z and A to Z but apart from E and D we get the message Invalid Command and we get disconnected from the server.
We try giving the D and E commands some arguments to see if their behavior changes.

No matter what additional input we give D the output format seems to stay the same although the string does change. The length of the encrypted data and IV are constant as well. Each time we use the D command we get a Key Reset message.

```
D
035b:3e4e9892b1ab053426dfa7168a0cc9827acf33d3480b
Key Reset!
D,
171f:73674b54f57f52874aed3027ece58d0f97780c88223e
Key Reset!
D,,,,
8e01:ba6b8ce2fa0e4d9f6b1888889df0110f4015b10a60fe
Key Reset!
D,AAAAA
a6ec:8f040a25951bdb727a974b7087c9cf733e44b427e2dd
Key Reset!
D,Test
b20f:0c823c92d80102913af7353254717d60b68f5e3d2a0e
Key Reset!
D
d4c0:b99815314cf8d5bec3d92be1204e66a044fc7050c48a
Key Reset!
D,BBBBBB
1026:bd9c781a6cd7d766040c0d539a13ebb9a590b1cebac1
Key Reset!
```

Next we try using the E command, it seems as though we need to use the command format E,<value> and anything else will give us an Invalid use of encrypt message and will return the usage message to us. When we supply a value to E, the output format mirrors the D command output, so it is likely that D is just performing the E command on the data stored in the service.

```
E.
Invalid use of encrypt. Usage E,<valuetoencrypt>
E,,,
Invalid use of encrypt. Usage E,<valuetoencrypt>
E,,,,,,
Invalid use of encrypt. Usage E,<valuetoencrypt>
E..
Invalid use of encrypt. Usage E,<valuetoencrypt>
E,
Invalid use of encrypt. Usage E,<valuetoencrypt>
E,A
fb51:9e
E,AAAA
78c9:f469f095
E,AAAA
9d16:001b2784
E,AAAAAAAA
dbfb:87a24590fd8ac117
```

```
E,TEST
1038:8283d469
E,TEST
7e3d:0c057546
E,1234567890
65e4:fdba65422ede91d82f10
```

When we use the E command, the output characters all seem to be within hex ranges 0-9 and a-f so it is likely that the strings are hex encoded characters. Additionally, we see no Key Reset messages so there is a good possibility that the key is not being reset between each use of the E function.

We also realise that each IV is different and the encrypted data length is always twice the length of the data we supply. Because the encrypted data length increases character by character this leads us to believe that it is a stream cipher we are dealing with rather than a block cipher.

We read into stream ciphers and stream cipher attacks and read that WEP had an issue due to the IV being only 24bits in size. We can see here that the IV only seems to be 2 hex encoded characters or 16bits in size. Potentially we could get reused keys and IVs which would lead to the reuse of a keystream, allowing us to decrypt the data returned from the D command. Lets test it!

We run the encrypt function with five A's 500 times then sort the results and count them, we end up with three IV collisions and for each of these collisions the response data is the same. This confirms that this program is reusing key and IV pairs and is vulnerable to this IV reuse attack.

```
#> for i in {0..500}; do echo "E,AAAAA" >> test.txt; done; echo "Q" >> test.txt
#The Q is so we get disconnected

#> tail test.txt
E,AAAAA
E,AAAAA
E,AAAAA
E,AAAAA
E,AAAAA
Q

#> cat test.txt | nc 172.16.1.20 1337 | sort | uniq -c | grep " [2-9] "
2 63b2:67c7b06d66
2 a315:beece8713b
2 f7db:08db5207b7
```

Now that we have confirmed that the server is vulnerable to an IV reuse attack we need to gather some results so we can recover the keystream and decrypt the service stored data.

We need to also keep in mind that because we aren't recovering the key, just the keystream, that we need to get at least as many response bytes as the output from the D command. To find the length we look at output from the D command and realise we need at least 22 bytes of keystream.

```
D
3868:3bff03fff1fe385eb6d68c9e84f6771564968d7b017d
```

We will write a script to encrypt 25 A's and store their results and then when we have 10000 unique IV's we will dump the flag and try to decrypt it.

```
import socket
clisock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
clisock.connect(("172.16.1.20",1337))
respdict = dict()
#Recv Key Reset message and Banner
print clisock.recv(1024)
print clisock.recv(1024)

while len(respdict) < 10000:
	#Send encrypt
	clisock.send("E,"+"A"*25+"\n")
	resp = clisock.recv(1024).strip()
	parts = resp.split(":")
	respdict[parts[0]] = parts[1]
	#print parts
	if len(respdict) % 200 == 0:
		print str(len(respdict))+"/10000"

#Dump the service secret
clisock.send("D\n")
secretdata = clisock.recv(1024).strip()
parts = secretdata.split(":")

#Look for the IV in the dict
if parts[0] in respdict:
	print "Found secret iv in dict. Dumping responses"
	print "   A's response:",respdict[parts[0]]
	print "Secret response:",parts[1]
else:
	print "Didn't find secret iv in dict. Try again"
```

We run this script against the server, wait a little while for IV's to be gathered and then we receive a message stating the dump command output has an IV matching one stored in our dictionary.

```
#> python solv.py
Connecting
```

```
Key Reset!

      Welcome to the FortCerts Certified Data Encryption Service
            This program uses very secure encryption
Commands:
E - Encrypt specified data
D - Dump service stored data
Output is in format <IV>:<Encrypted Data>

200/10000
**** SNIP (about 7 minutes) ****
9600/10000
9800/10000
10000/10000
Found secret iv in dict. Dumping responses
   A's response: 32fe79d33e7055190ec47c1778c1029946c9fc47398c613936
Secret response: 35cd51f711555a373bec5e337be52fbe66fbc9334af8
```

Now that we have our responses with the reused keystream we first need to recover the keystream. To do this we need to keep in mind that stream ciphers generate a keystream which is then xored byte by byte against the plaintext to generate the ciphertext. We know the plaintext for the A's response so we can xor the response data against the plaintext to get the actual key stream.

```
    A's response: 32fe79d33e7055190ec47c1778c1029946c9fc47398c613936
   A's plaintest: 41414141414141414141414141414141414141414141414141
Shared keystream: 73bf38927f3114584f853d56398043d80788bd0678cd207877
```

Now that we have recovered the shared keystream we can use it to decrypt the Dump command response and retrieve our flag **FriendNoticeBelfast525**.

```
   Dump response: 35cd51f711555a373bec5e337be52fbe66fbc9334af8
Shared keystream: 73bf38927f3114584f853d56398043d80788bd0678cd207877
  Dump plaintext: 467269656e644e6f7469636542656c66617374353235
Plaintext(ASCII): FriendNoticeBelfast525
```